TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

HENRIK PAAVILAINEN

SAAS CONFIGURATIONS AND CUSTOMIZATIONS: MANAGEMENT TOOL FOR DIGITAL SIGNATURE SERVICE

Master of Science Thesis

Examiner: Professor Hannu-Matti Järvinen

Examiner and topic approved on 29.3 2017

ABSTRACT

PAAVILAINEN, HENRIK: SaaS configurations and customizations: management tool for digital signature service
Tampere University of Technology
Master of Science Thesis, 57 pages
August 2017
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Hannu-Matti Järvinen

Keywords: SaaS, cloud computing, cloud services, web development, multi-tenancy, customizability, configuration, customization

Today, cloud computing – a result of combining existing technologies – is a popular paradigm that has brought many benefits for users and enterprises. Cloud computing fosters the provision and use of IT infrastructure, platforms, and applications of any kind in the form of services that are available on the Web. Expensive initial hardware and software investments are not necessary anymore as the resources can be acquired as a service from cloud providers with a pay-per-use pricing model. One aspect that cannot be overlooked in cloud computing is multi-tenancy. It is a property of a system where multiple customers, so-called tenants, transparently share the system's resources. It leverages economies of scale where users and cloud providers benefit from reduced costs, which is a result of higher system density and increased utilization rate of resources. This model surpasses the traditional methods of using single-tenant architecture and ASP model in which a single instance or server is provisioned solely for one customer.

Customizability is an essential part of multi-tenant systems. Ideally cloud application vendors wish that every user would be satisfied with the standardized offering, but usually users have their own unique business needs. Customizability can be divided into configuration, which supports differentiation by pre-defined scope, and customization, which supports tenant's custom code. Software variations can be applied to user interface, business logic related workflows, underlying data and reporting utilities. Multi-tenancy shares a lot in common with software product line engineering. However, implementing multi-tenancy and supporting differentiation between tenants have to be carefully planned. Increased complexity may have an impact in maintenance costs and re-engineering costs can be significant.

Goal of the thesis is to first examine the requirements for a multi-tenant application, and based on the research, to develop a prototype of a configuration management tool in order to solve the customization need produced by tenants' unique business requirements. The target environment consists of a new SaaS service called SignHero, which is a digital signature service suited for companies that want to shift their signing process to modern times. The scope includes three variability points: customizing the logo in the signing page, customizing the logo in the emails and saving a default workflow. The developed tool fulfills the requirements, and the main service was extended to apply the saved configurations. The implementation leaves many improvement possibilities related to customizability and cloud characteristics. Findings promote the fact that customizability has to be initially included in the product design.

TIIVISTELMÄ

PAAVILAINEN, HENRIK: SaaS-palvelun konfigurointi ja kustomointi: konfiguroinninhallintatyökalu digitaaliselle allekirjoituspalvelulle
Tampereen teknillinen yliopisto
Diplomityö, 57 sivua
Elokuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: SaaS, pilvilaskenta, pilvipalvelut, web-kehitys, multitenantti-arkkitehtuuri, kustomoitavuus, konfigurointi, kustomointi

Tänä päivänä pilvilaskenta – monien jo ennestään olemassa olevien teknologioiden yhdistelmä – on suosittu paradigma, joka on tuonut monia hyötyjä niin käyttäjille kuin yrityksillekin. Pilvilaskenta edistää minkä tahansa IT-infrastruktuurin, alustojen ja sovellusten tarjoamisen palveluina verkossa. Kalliiden laitteistojen ja sovellusten investoinnit eivät ole enää tarpeellisia, koska resurssit saa hankittua palveluna pilvipalveluiden tarjoajilta käyttöön perustuvalla hinnoittelulla. Eräs tärkeä järjestelmien ominaisuus on multitenantti-arkkitehtuuri, jonka avulla useat asiakkaat jakavat huomaamatta saman systeemin resurssit keskenään. Tämä mahdollistaa mittakaavaedun, jossa käyttäjät sekä pilvipalveluiden tarjoajat hyötyvät vähentyneistä kustannuksista, mikä on seurausta asiakasmäärästä ja resurssien paremmasta käyttösuhteesta. Tämä malli korvaa perinteisen singletenantti-arkkitehtuuri- ja sovellusvuokrausmallin, joissa yksi instanssi tai palvelin on tarkoitettu vain yhdelle asiakkaalle.

Kustomoitavuus on olennainen osa multitenantteja järjestelmiä. Ihanteellisesti pilvipalveluiden tarjoajat toivovat jokaisen käyttäjän olevan tyytyväisiä sovellukseen sellaisenaan, mutta useimmiten käyttäjillä on omat yksilölliset tarpeensa. Kustomoitavuus voidaan jakaa konfigurointiin, joka sallii erilaistamisen ennalta määritetyn laajuuden mukaan, ja kustomointiin, joka sallii asiakkaan oman koodin suorittamisen. Sovelluksen variaatiot voivat koskea käyttöliittymää, liiketoimintalogiikkaa, datakerrosta ja raportointiominaisuuksia. Multitenantti-arkkitehtuurilla on paljon yhteistä tuoterunkoon perustuvan kehityksen kanssa. Multitenantin järjestelmän toteuttaminen ja erilaistamisen tukeminen asiakkaille täytyy kuitenkin suunnitella huolellisesti. Monimutkaisuuden kasvamisella saattaa olla suuri vaikutus sekä toteutus- että ylläpitokustannuksiin.

Työn tavoitteena on selvittää edellytykset multi-tenantin sovelluksen toteuttamiselle, ja kehittää tehdyn tutkimuksen perusteella konfiguroinninhallintatyökalu ratkaisemaan asiakkaiden yksilöllisten vaatimusten aiheuttama tarve kustomoitavuudelle. Kohdeympäristö koostuu uudesta SignHero-nimisestä SaaS-palvelusta, joka on uudistushenkisille yrityksille suunnattu sähköinen allekirjoituspalvelu. Työ kattaa kolme muuntelukohdetta: logon kustomointi allekirjoitussivulla, logon kustomointi sähköposteissa, ja oletusarvoisen työnkulun tallentamisen. Toteutettu työkalu täyttää alkuperäiset vaatimukset, ja hallinnointityökalun kautta tallennetut konfiguroinnit otettiin myös käyttöön itse palvelussa. Toteutettu työkalu kuitenkin sisältää monia parannuskohteita liittyen kustomointiin ja pilvipalvelun ominaisuuksiin. Löydökset tukevat ajatusta siitä, että kustomoitavuuden tulisi olla olennainen osa tuotteen suunnittelua.

PREFACE

This thesis was done for Avaintec as a part of cloudification project called SignHero.

I want to express my gratitude to Avaintec for granting me the opportunity to do this thesis for the company. Big thanks to my instructor Petteri Sulonen who was always willing to help and share his knowledge, and to professor Hannu-Matti Järvinen for guiding through the thesis. I would also want to thank my girlfriend, family and friends for the support. Special thanks to Mari Torikka for motivational words and all the help related to my varying graduation concerns.

Münich, 23.8.2017

Henrik Paavilainen

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

NIST	National Institute of Standards and Technology
CAPEX	Capital Expenditures
OPEX	Operational Expenditures
SLA	Service Level Agreement
QoS	Quality of Service
SaaS	Software as a Service
PaaS	Platform as a Service
IaaS	Infrastructure as a Service
API	Application Programming Interface
ASP	Application Service Provider
CRM	Customer Relationship Management
VM	Virtual Machine
VMM	Virtual Machine Monitor
MTA	Multi-Tenant Application
SPA	Single-Page Application
DOM	Document Object Model
AJAX	Asynchronous Javascript and XML
JSON	Javascript Object Notation
UI	User Interface
SCM	Software Configuration Management
SPL	Software Product Line
UDD	Universal Data Dictionary
REST	Representational State Transfer
SOA	Service-Oriented Architecture
DevOps	Development Operations
XWFM	X-Web Forms
XDSS	X-Digital Signature Suite
AMD	Asynchronous Module Definition

1. INTRODUCTION

Nowadays applications and computing resources are always available in the cloud, ready to be provisioned to the users whenever needed. For the past decade the popularity of cloud computing has been increasing because of all the benefits that concern end users, developers as well as cloud providers. Expensive, on-premise hardware investments are history as the whole infrastructure is outsourced to a cloud provider who keeps the services highly available by handling the upkeep and scaling. Users usually pay for what they actually use, besting the option of preserving and paying for lots of computing resources regardless of actual usage.

Cloud providers try to cope with this situation of having unused resources by increasing system density, i.e. sharing computing resources with as much customers as possible. In fact, resource sharing and so called multi-tenancy can be considered as one of the most important aspects in cloud computing, because leveraging economies of scale benefits both the provider and users.

Multi-tenancy can be examined on various levels, not just hardware. Cost-wise, application instance turns out to be the most efficient level to implement multi-tenancy. Hardware and software resources are cost-efficiently divided across customers, and the general overhead is minimal, hence lowering the running costs of the application. Ideally cloud application vendors would want that every user would be content with the standardized offering, but this is usually not the case because each user has their own unique business needs. This generates requirements to provide tools that allow customers to easily shape the service to their liking, which may include modifying the appearance or underlying data sets of application.

The aim of the thesis is to first study the topic of multi-tenancy and customizability in order to find out the requirements of a multi-tenant application. The second goal is to implement a prototype of a configuration management tool – based on the research – for a new cloud service called SignHero, and modifying the service to perform accordingly to the saved configurations. SignHero, developed by Avaintec, is a digital signature service that makes possible for companies to digitally sign and archive PDF documents.

Chapter 2 familiarizes the reader with cloud computing basics. The definition of cloud computing is introduced alongside with different types of cloud. The chapter continues with explaining virtualization technologies, resource sharing concepts and multi-tenancy. In the end web application is explained for its important role in cloud popularity. Chapter 3 focuses on differentiation concepts and challenges of multi-tenant applications, and gives an example of a highly customizable application called Salesforce.com. Design principles of a multi-tenant cloud are presented in Chapter 4. Chapter 5 intro-

duces SignHero products and the requirements for the configuration management tool, along with technologies relevant to the work. Chapter 6 goes through the implementation of the configuration management tool. The conclusions are presented in Chapter 7.

2. CLOUD COMPUTING BACKGROUND

Making business has been revolutionized by cloud computing. There are many analogies from renting a car, or using electricity or tap water: the idea is that there is a bigger pool of resources maintained by someone else, and the resources are taken and used whenever needed (Fehling et al. 2014; Antapoulos & Gillam 2010). However, the definition of cloud has become blurred due to adding the term "cloud" on pre-existing products on a field where offering IT resources over a network is nothing new (Fehling et al. 2014).

In this chapter the background theory of cloud computing is described. Section 2.1 introduces the overview of cloud computing and the important aspects based on the NIST definition, which can be considered as the prevalent definition on the field (Fehling et al. 2014). Section 2.2 introduces service and deployment models. Section 2.3 discusses practical implementation of cloud characteristics by introducing virtualization and containers, where the latter is gaining more footing in today's cloud computing. More comprehensive resource sharing concepts and this thesis' main topic multi-tenancy are described in Section 2.4. Section 2.5 introduces web applications which are an essential part of boosting the popularity of cloud computing.

2.1 Definition of cloud computing

Cloud computing fosters the provision and use of IT infrastructure, platforms, and applications of any kind in the form of services that are available on the Web. These services are then provided on demand to consumers and billed on a usage-basis. As cloud computing is still a relatively new paradigm, there are multiple definitions and interpretations of the term available. To ease the discussion around the topic, the National Institute of Standards and Technology (NIST) in the U.S. has created a commonly-quoted definition specifying the essential characteristics alongside the different service and deployment models. (Baun et al. 2011)

The NIST (Mell & Grance 2011) defines cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction". The five fundamental characteristics of the cloud model are:

- On-demand self-service: Resources, such as server time and network storage, can be automatically provisioned by the cloud consumer when needed.
- Broad network access: Services are available over the network by any device.

- Resource pooling: Physical and virtual resources are pooled to serve multiple consumers using a multi-tenant model. Resources are adjusted (assigned and re-assigned) to the users' actual demand.
- Rapid elasticity: Computing capabilities appear infinite by (automatic) scaling rapidly outward and inward.
- Measured service: Resource usage is monitored, controlled, reported to enable usage-based billing.

Fehling et al. (2014) note that these cloud characteristics may be known already and are often used in different well-established products and services, such as server hosting solutions or public Web applications. Even the technologies behind cloud computing are not entirely new: cloud computing can be seen as "the result of many years of evolution dating back to the first computers" (Kavis 2014), adopting many ideas from utility computing, using concepts of parallel and distributed systems, and relying heavily on virtualization (Marinescu 2013; Kavis 2014).

What has been done previously on the mainframe can now be done fast and at scale, allowing the consumer to pay for the resources only when they are needed without ever buying any hardware. Computing resources are offered as services over the Internet which allow more flexibility, speeding up time-to-market and making it possible for the consumer to focus on business issues and spending less time managing infrastructure. This has an impact on the economic scale by lowering the initial investment. (Kavis 2014)

Fehling et al. (2014) state regarding the economic benefits that for cloud customer cloud computing have enabled the shift of capital expenditures (CAPEX) in IT to operational expenditures (OPEX) as long-term upfront investments in IT resources have been reduced. This allows more flexibility for the business to increase or decrease the ongoing costs based on the growth, allowing customer to benefit from measured service property and its pay-per-use billing. For cloud provider, rapid elasticity and resource pooling properties allow taking advantage of *economies of scale* if sharing of resources is done efficiently. In cloud computing, economies of scale means that offering a cloud service to a very large number of customers reduces the costs for individual customers.

Cloud service is supposed to be highly available, accessible around the clock for the users. Vendors include a Service Level Agreement (SLA), which specifies in detail the availability of the service and also possible compensations if the agreed service levels cannot be reached. SLA specifies Quality of Service (QoS) and it is a mutual agreement between both the service provider and the service consumer with respect to security, priorities, responsibilities, guarantees, and billing modalities. It also includes metrics for e.g. throughput and response times. From business point of view, service and its quality can be differentiated by offering different kinds of quality levels, for example Basic and Premium models. (Baun et al. 2011)

2.2 Service and deployment models

The NIST definition includes three service models and four deployment models. The service models are Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). In SaaS, the consumer can use the provider's application deployed to the cloud through a browser or a program interface with no need for the knowledge about the underlying cloud infrastructure. PaaS allows the consumer to create a cloud application with the tools issued by the platform provider, but without control over e.g. the network, servers or storage. IaaS is an environment where arbitrary software can be deployed and run by the customer, while also offering more control over operating systems, storage and deployed applications. (Mell & Grance 2011)

All three service models can be deployed in a public, private, hybrid or community cloud. Public cloud is meant to be used by the general public and is owned, managed and operated by a business, academic or government organization (Mell & Grance 2011). According to Kavis (2014), essentially a public cloud is about outsourcing data center and its infrastructure management to a third party, knowing that there are other customers using the same seemingly endless resources. Public cloud is impaired by the lack of control, regulatory issues and configuration possibilities.

Private cloud is provisioned for exclusive use by a single organization. It can be managed by the organization itself as an on-premise solution, or hosted elsewhere while being managed by a third party (Mell & Grance 2011). Kavis (2014) states that both options solve some of the mentioned drawbacks of public cloud, but the end user makes a trade-off decision by giving up core advantages such as rapid elasticity, resource pooling and pay-per-use pricing. User has already paid for the infrastructure whether the resources are utilized or not. Despite the downsides, according to Marinescu (2013), private clouds offer a cost-effective alternative to public clouds. Essentially private cloud is built using the same structural components as a commercial one. There are both proprietary and open-source cloud control infrastructures available, notably Microsoft offering the first one (Microsoft 2017) and OpenStack the latter (OpenStack 2017).

In community cloud the infrastructure is shared by a specific community of consumers, and it can be owned and managed by one or more members of the community or it can be operated by a third party. Hybrid cloud combines two or more cloud types bounded together by standardized or proprietary technology. (Mell & Grance 2011)

Kavis (2014) describes that SaaS, on top of the application stack, is the most widely used of the three service models. SaaS is a complete application delivered to the end user. Vendors take care of the entire infrastructure and its maintenance, providing security updates and patches, database recoveries, and feature updates seamlessly to the consumers without downtime. Consumer has to only take care of user management and application-specific configurations. Web browsers are the most common way to use the

service, but often SaaS vendors provide an *Application Programming Interface* (API) in order to integrate the SaaS into customers' existing applications. SaaS is similar to traditional *Application Service Provider* (ASP) model, but Rountree & Castrillo (2013) list some notable differences: ASP typically manages one dedicated application for each customer while SaaS usually uses one shared instance between customers, and instead of using Web-based application like in SaaS, the application was hosted on the client server. Walraven et al. (2011) state that in SaaS the economies of scale has a more important role than in ASP model.

Marinescu (2013) depicts that SaaS suits well enterprise services such as workflow management, communications, Customer Relationship Management (CRM), digital signatures, including occasions where there is a short-term need or a significant peak in demand. On the other hand, SaaS cannot be recommended for applications that require real-time response, nor use cases where data is not allowed to be stored externally. Rountree & Castrillo (2013) further explains that the latter may also cause issues in reporting and business intelligence if the service lacks integration possibilities, and there is no direct access available to the data.

PaaS is a service offering which provides a platform for customers and their computing needs. Usually PaaS is used for development purposes, because it offers a fast way to develop a Web application directly to the cloud. The vendor takes care of the underlying infrastructure, and offers a wide range of tools and services, including support for different programming languages, in the platform itself to make building and deploying applications effective. Organizations might find it tempting to use a PaaS and develop the application themselves, if there is an urge to move towards public cloud model and there are no suitable applications available on the market as a public SaaS. The provider does not generally have any control on how the application or service is developed, nor does not take care of the application's maintenance and updates. What PaaS vendor is responsible for is everything at the development platform level and below, which includes for example keeping the operating system up-to-date. (Rountree & Castrillo 2013)

Concept of PaaS is effective, but it still has some concerns and challenges. One of the downsides of a PaaS is so called *vendor lock-in*, where users might become too dependent on a particular PaaS vendor, which makes it difficult, if not impossible, to move the application to another platform (Baun et al. 2011). Marinescu (2013) notes that PaaS may not be particularly useful if propriety programming languages are used or application needs to customize the underlying hardware or software to increase performance. Buyya et al. (2011) state that there might be some security concerns because the PaaS provider has the control over the platform and has a direct access to all of the applications and data. Hence, large enterprises have taken the best use of hybrid clouds to keep the critical data in on-premises.

On the bottom of the cloud stack is IaaS cloud computing delivery model. It is a service with the most extensive management possibilities for the deployed software, which can be e.g. whole operating systems, storage, Web servers, virtual instances or applications. The consumer has an access to all the resources that are needed to run the software, but cannot manage or control the underlying cloud infrastructure. (Marinescu 2013)

Typical for IaaS is that it supports dynamic scaling, underlying hardware is shared among multiple users and it has variable costs with utility-based pricing model (Marinescu 2013). According to Fehling et al. (2014), IaaS often has an on-demand self-service portal allowing customers to configure and provision servers, storage and network connectivity, with the possibility to monitor the resource consumption and corresponding charges. *Rapid elasticity* of IaaS makes it possible to respond to actual usage by adding or removing infrastructure resources within minutes, and it is one of the key factors in making IaaS clouds successful and widely accepted. As stated by Marinescu (2013), IaaS is also useful in cases where the demand for computing resources is erratic. There might be a resource need originated from fast business growth, the resource peak is generated by business growth and there is no urge to invest in own computing infrastructure. Regarding administrative costs, it is good to keep in mind that IaaS incurs costs similar to a traditional computing infrastructure.

Kavis (2014) emphasizes the importance of managers and architects fully understanding the benefits and disadvantages that cloud computing comprises, and what are the roles and possibilities of each service and deployment model. When leveraged properly, an organization can benefit from unprecedented agility and greatly reduced costs. However, if cloud computing is not fully understood, an organization may end up with a software solution that turns into a burden, never delivering its promises to the business. Moving legacy applications to the cloud also require in-depth planning because of the drastically different target system.

2.3 Virtualization and containers

Virtualization is one of the underlying techniques that is often utilized to enable cloud characteristics in cloud infrastructures. Majority of the cloud providers have taken it into use because it supports resource pooling, rapid elasticity and increases utilization rate of the physical hardware (Rountree & Castrillo 2013; Omote et al. 2015) There are many types of virtualization, such as hardware, network and operating-system virtualizations. (Rountree & Castrillo 2013)

As described by Marinescu (2013), virtualization, dating back to 1960s, allows creating virtual resources from actual physical resources: it abstracts the resources and simplifies their use, allows replication and isolates users from one another. The most common type is hardware virtualization, in which a simulated physical system, a *virtual machine* (VM), is created. VM can be run on top of an actual system, and in most of cases, there

are more simulated systems running on a single computer in order to increase system density. A *Virtual Machine Monitor* (VMM), also called as a *hypervisor*, is software that handles partitioning resources into virtual machines. Partitioning is done securely, and it isolates the users from each other while making sure that resource utilization of one user does not exceed the limits. According to Buyya et al. (2011), VM provisioning and migration plays an important role in IaaS model in terms of service's elasticity. Provisioning means creating a new virtual server and its configurations based on some specific requirements, which can be done either cloning an existing VM or by a template. Provisioning can be automated, for example *Amazon Auto Scaling* can be set up and used in *Amazon EC2* environment for scaling the service seamlessly. Migration is the movement of VM from one host server to another on the fly. Migration and replication together are important in achieving high availability and performance, and in load balancing, adjusting to the demand can be done by assigning more resources to one host or move one highly utilized to underutilized host. Provisioning and migrating a new VM is a matter of minutes, thus playing a big role in achieving SLA agreements and QoS specifications.

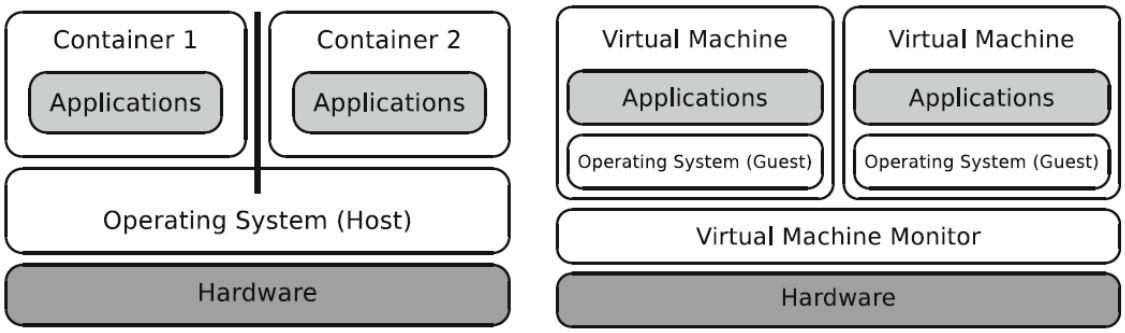

Figure 2.1. OS based virtualization and hypervisor (Type-1) based virtualization (Baun et al. 2011)

Previously introduced hypervisor-based virtualization is not the only type of hardware virtualization. On the left of Figure 2.1 is shown an alternative, *operating system level virtualization*, exposing the different layers of virtualization next to a Type-1 hypervisor. Type-1 means that the hypervisor runs directly on the hardware whereas in Type-2 the hypervisor runs on operating system (Baun et al. 2011). Marinescu (2013) explains the differences between aforementioned virtualization concepts by stating that operating system level virtualization does not utilize VMMs or VMs but instead independent *containers*, isolated operating system instances, on the host machine. These kinds of virtualization systems have performance advantages over VMM-based systems, but are also subject to several constrains, for example *OpenVZ* requires that both the host and the guest OS are Linux distributions. According to Nigam (2015), there are also *application containers* which cannot have their own OS. This leaner version of container makes them highly useful in sense of system density and startup times. Number of containers can go up to hundreds, and by contrast the same server could host only a few dozen VMs. One example of this kind of a container is *Docker* (Coleman 2016).

There are also alternatives to hardware virtualization. Omote et al. (2015) write that there is a new, emerging type of IaaS, so called *bare-metal cloud*, that uses physical machines instead of VMs. Even though VMs provide reasonable performance for most of the applications, they lack top-notch performance. This has led to use bare-metal clouds in certain type of resource intensive applications that have significant performance, functionality and security demands.

Marinescu (2013) notes that there are two distinguishable scaling models that are used to make the service elastic: *vertical* and *horizontal*. In terms of virtualization techniques, vertical scaling increases the amount of resources allocated to each VM by for example increasing their share of the CPU time or by migrating the instances to more powerful servers. The latter has some extra overhead as the VM is first stopped for a snapshot, then transported to new location to be started again. In *modularly divisible applications* this type is the only option. Vertical scaling is also known as *scaling up*, and typically it does not require software changes to leverage the new infrastructure as long as it is of same infrastructure type (Kavis, 2014). According to Marinescu (2013), more common type to handle varying workload is to use horizontal scaling. This method increases the total number of VMs during peak times, and accordingly reduces the number of VMs when the workload is lower. To make this setup work, a load balancer is needed to handle the traffic and to distribute the incoming requests to the multiple VMs. Kavis (2014) state that horizontal scaling can be accomplished by adding more infrastructure that runs in conjunction with the existing one, and it can be done at multiple layers of the architecture. Horizontal scaling can also be called as *scaling out*.

2.4 Resource sharing concepts and multi-tenancy

In traditional single-tenancy architecture a single instance of a software application and supporting infrastructure is used by one customer as a standalone application (Saraswathi & Bhuvaneswari 2014). This is not relevant anymore because in cloud computing resource sharing can be considered as one of the most important aspects. Virtualization, probably today's most widely adopted sharing approach, provides an easy way to share a single server, and it can be considered as the first step towards efficient operation (Krebs et al. 2012). In IaaS, cloud provider profits from multi-tenancy because e.g. virtualization allows having many customers on the same hardware, and the gained system density may reflect reduced costs for the customer (Rountree & Castrillo 2013). According to Kabbedijk et al. (2015), the topic of multi-tenancy appeared relatively recently in the domain of software and hardware systems, as the first appearance can be dated to 2006 (Chong & Carraro 2006) when Chong and Carraro explicitly mentioned multi-tenancy in the paper "Architecture Strategies for Catching the Long Tail".

Kabbedijk et al. (2015) define multi-tenancy after the extensive research around the topic in academic and industrial world as following: "Multi-tenancy is a property of a

system where multiple customers, so-called tenants, transparently share the system's resources, such as services, applications, databases, or hardware, with the aim of lowering costs, while still being able to exclusively configure the system to the needs of the tenant."

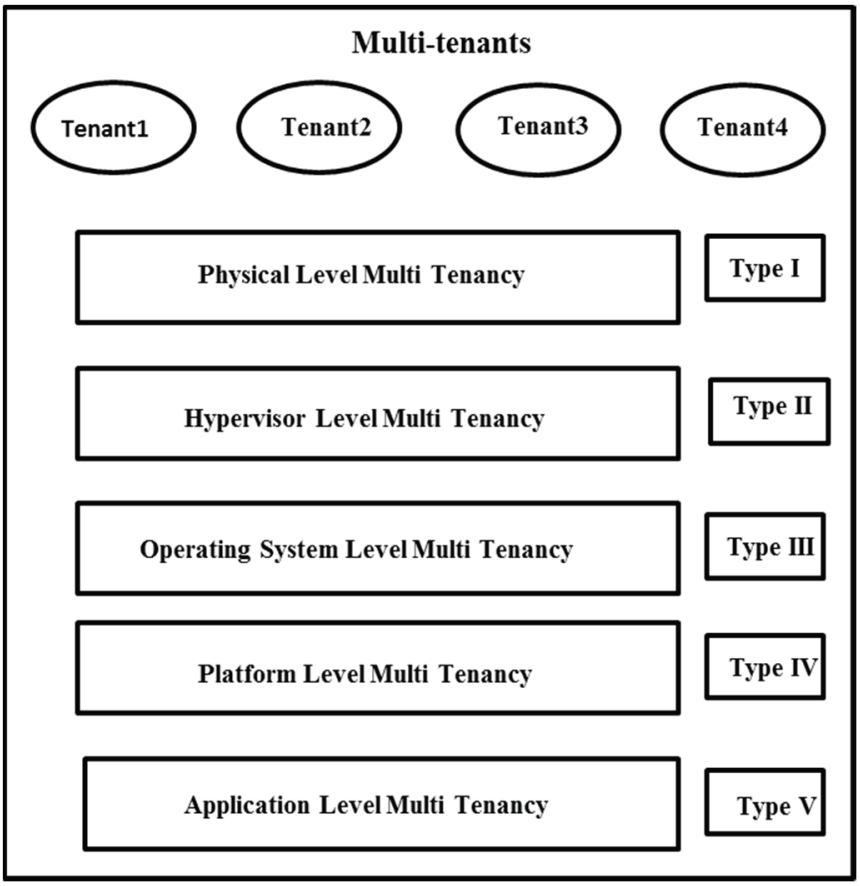

Figure 2.2. Multi-Tenancy Levels. (Ashalatha & Jayashree 2016)

Figure 2.2 shows the overview of resource sharing concepts and introduces the various types of multi-tenancy levels. Each approach has their own trade-offs between minimizing operational costs, minimizing upfront application (re-)engineering costs, and maximizing flexibility to respond to varying customer requirements (Walraven et al. 2011). Multi-tenancy levels are explained by Ashalatha & Jayashree (2016). *Physical level* multi-tenancy means sharing the data center and using dedicated hardware for each tenant thus providing better performance than virtualization, but is hindered by poor scalability and high operational costs. *Hypervisor level* is suited for non-multitenant applications and it offers better scalability. In *operating system level* multi-tenancy, the database and application server is shared. *Platform level multi-tenancy* consists of sharing hardware, operating system and application server among the users. *Application level* multi-tenancy means that single application instance is shared by all the tenants. First four types are used when the application has no built-in support for multi-tenancy. According to Walraven et al. (2011), resource sharing is most efficient on the level of application instances. This approach greatly reduces the operational costs: hardware and

software resources can be more cost-efficiently divided and multiplexed across customers, and the overall maintenance is significantly simplified because upgrading the application software affects all tenants at once. Other benefits regarding to Krebs et al. (2012) is that the general overhead is minimal and application level multi-tenancy allows performing explicit workload and resource management with e.g. requests and threads.

Krebs et al. (2012) describe that an application that is designed to serve multiple customers (tenants) with a single, configurable instance is referred as *multi-tenant application* (MTA). At this application-level multi-tenancy, a tenant can be defined as a group of users that share the same view on an application. This view consists of the data they access, configurations, the user management, particular functionality and related non-functional properties. Usually the groups are members of different legal entities and therefore there are some restrictions regarding e.g. data security and privacy. Transparency mentioned in the definition Kabbedijk et al. (2015) refers to hiding the presence of tenants from each other, separating multi-tenant system from *multi-user* applications such as Facebook. Distinction between multi-user and multi-tenant systems is also made by Bezemer & Zaidman (2010), stating that multi-user application has only minor configuration possibilities, and there are rarely differences in the SLAs among the tenants. Baun et al. (2011) emphasize the isolation and security properties of a multi-tenant system by noting that it is crucial the tenants are isolated from each other, so they cannot see other tenants' data.

Regarding true multi-tenancy, Krebs et al. (2012) suggest that these *multi-instance* called solutions that share a data center or a middleware, or rely on virtualization cannot be considered to be part of real multi-tenancy. They may provide multitenant-like behavior for the tenant, but most importantly they lack in sharing and in efficiency, and therefore multi-instance cannot be considered as the first choice in cloud development. In addition, the development of such applications cannot be considered as anything special. Nevertheless, there are still cases where using one instance per tenant is a viable option. Sharda et al. (2014) point out that these reasons may be related to e.g. lower time-to-market or application's data isolation properties. It is easier to move existing applications to cloud thus reducing time-to-market, and separate instances allow easier approach to tenant-specific customizations. Also isolation among tenants is strong, tenants rarely affect each other and the risk of data leakage is low. This approach prevents from benefitting of economies of scale, and application upgrade cycles will end up being costly and error prone. Moreover, Bezemer & Zaidman (2010) note that using instance-per-tenant approach is better if the number of tenants is likely to remain low so the cloud provider does not suffer from the ever increasing maintenance costs.

SaaS provider needs to consider carefully whether to add support for customization of the service because it may turn up to be costly for both the provider and in the end also for the customer (Rountree & Castrillo 2013). For existing SaaS applications that are

expecting to take a step into multitenancy, Momm & Krebs (2011) introduce a simple cost model for decision-makers to find the optimal multi-tenancy approach for the application. The cost model is for calculating the breakpoint where achieved cost savings have amortized the costs resulted by re-engineering activity. The cost model can be expressed as formula

$$NumMonthToBreakEven = \frac{Initial\ Reeng.Costs}{\Delta Operating\ Costs} \quad (1)$$

where *Initial Reeng. Costs* is the initial costs for implementing activities that are required to be done once for the chosen multi-tenancy option, and Δ*Operating Costs* is the monthly cost savings which are acquired after taking multi-tenancy into use. Monthly operating costs include fixed costs per instance – e.g. application, middleware or hardware – as well as costs per tenant. In the long run, the variant with lowest incremental cost is always the best. Making the decision involves planning the life time of the service to determine if the investment pays off in the estimated service life time.

Using the function (1) requires calculating the operating costs, which include monthly costs for operating system, middleware and application instances. Operating cost functions for each multi-tenancy implementation option can be expressed as formulas

$$OpCost_{shInfra} = CostOS_{Base} + n * (CostMW_{Base} + CostApp_{Base} + CostApp_{Tenant})$$

$$OpCost_{shMW} = CostOS_{Base} + CostMW_{base} + n * (CostApp_{Base} + CostApp_{Tenant})$$

$$OPCost_{shApp} = CostOS_{Base} + CostMW_{Base} + CostApp_{Base} + n * CostApp_{Tenant}$$

where $CostOS_{Base}$ is the base cost for operating system, $\mathrm{CostMW_{Base}}$ is the base cost for middleware instance, $CostApp_{Base}$ is the base cost for application instance, $CostApp_{Tenant}$ is the cost for tenant-specific application efforts, and *n* is the number of expected number of tenants. Costs for resources that are not shared are multiplied with the number of tenants. Cost functions are suitable for comparing different options, even though these functions might not be linear in reality. Using these functions requires estimating the effort and the maintenance costs of single components for each alternative. Estimation can be done with techniques such as *Function point analysis* or *Architecture-Centric Project Management.*

2.5 Web applications

One of the main drivers making cloud applications, and most notably SaaS based applications, increasingly popular are the improvements in Web-based applications. Web applications are accessed via the Internet and generally do not require any other client installations than a Web browser which acts as the execution environment. Lately look-and-feel, ease of development and overall quality has increased drastically. Wide varie-

ty of tools, for example HTML5, Javascript, CSS and server side frameworks, can be used for creating robust Web applications. Quality and ease-of-use combined with the ability of cloud services being available from anywhere and typically from any device makes web applications compelling in cloud-based scenarios. It can also be argued that web applications are becoming the de facto standard for offering applications. (Rountree & Castrillo 2013)

According to Fink & Flatow (2014), popular way to implement a modern web application is making it as a *single-page application* (SPA). When making the initial request to the site, only one page is served along with initially needed HTML, Javascript and CSS. The notable chance compared to the traditional way of requesting new HTML pages when navigating on the web page is that an SPA handles the transitions by making queries in the background to the server and based on the response, refreshing the content on the single page with Javascript. Usually the Javascript logic involves using the page's *Document Object Model* (DOM), which an interface to access and update the content, structure and style of documents (W3 2009).

In a SPA, there are no full refreshes of a single web page, and the state can be persisted in the browser memory. The technique used in the background is called *Asynchronous Javascript and XML* (AJAX): the back-end server has an API which is targetted by asynchronous interactions, and responses are often – despite the name – in *Javascript Object Notation* (JSON) instead of XML. Whichever the chosen data format is, client side handles moving from one page to another by front-end routing and templating mechanism. Business logic is performed on the client side, which leaves the server to handle authentication, validation or persistency to databases. Because of this, “SPAs are super responsive and they make users feel like they are using native application”. (Fink & Flatow 2014)

3. DIFFERENTIATION IN MULTI-TENANT APPLICATIONS

Sun et al. (2008) describe that SaaS is about serving customers in a large scale by delivering software functionalities over Web. The most ideal case for SaaS vendors is that every client would feel comfortable using a completely standardized offering. However, this is usually not the case as every client has their own unique business needs. In multi-tenant environments, this generates requirements for the software to provide tools that allow customers to shape the service easily. Sharda et al. (2014) note that in order to achieve customizability and extensibility for each tenant, the application must be partitioned into base and variable parts. The base parts will handle the instantiation of tenant-specific objects at runtime, making it possible to have a personalized user interface (UI), workflows, business logic and data fields.

Sun et al. (2008) mention that creating widely customizable software is not a new problem. Many academic research and industrial best practices, for example Software Configuration Management (SCM) theory developed by Roger Pressman through software engineering research, have addressed this issue. Traditional on-premise software such as SAP enterprise software enables customizability by providing own tools and script-based programming tool called ABAP. Also Krebs et al. (2012) states that "developing a widely configurable software instead of customer specific branches is a question related to *product line engineering* and not specific for MTAs".

A customizable SaaS application has been defined multiple ways in literature. The following definition is from Software as a Service: Configuration and Customization perspectives by Sun et al. (2008), which introduces two major approaches to provide a tailored SaaS Service: *Configuration* and *Customization*. These two concepts used in MTAs are defined in Sections 3.1 and 3.2. Section 3.3 gives an insight about variability and its types in applications. Section 3.4 introduces a competency model and strategies that can be used when designing an adjustable application. Section 3.5 gives an insight how industrial leader Salesforce.com has solved this issue, and finally the MTA maintenance and challenges are described in Section 3.6.

3.1 Configuration

The definition of configuration varies a lot depending on the context. In studies by Sun et al. (2008) and Walraven et al. (2011), configuration does not involve source code change of the SaaS application, but rather support variance through setting pre-defined parameters or leveraging tools to change the application functions within pre-defined scope. On the other hand, configuration can also been seen as a subtype of customiza-

tion (Krebs et al. 2012; Tsai & Sun 2013). Krebs et al. (2012) consider that the key enabler for MTAs is the application's ability to handle different tenant specific configurations such as UI, the system functional/non-functional behavior and the services.

Because the application instance on a single server serves multiple users from different organizations, writing custom code to implement tailored end-user experience is not possible due to the changes affecting all the others, too. Configuring the application through tenant-provided metadata makes it challenging for the SaaS architect to ensure the task of configuring is simple and convenient for the customer. The architect must bear in mind that each configuration should not cause extra development or operation costs. This emphasizes the importance of good UI design practices, and there should be as much effort put into designing the interface for the end user as in the underlying configuration interfaces. For example, the screens should be simple and intuitive, presenting all available options without causing information overload, and make it distinguishable to detect what can and what cannot be changed within a given scope. (Chong & Carraro 2006)

When speaking of customization in the scope of SaaS applications, it usually means using the configuration approach (Kabbedijk et al. 2015). Nevertheless, configuration should be on a higher priority compared to customization; the configurable limit should be extended as far as possible towards clients' unique requirements (Sun et al. 2008).

3.2 Customization

Customization is an alternative to implement differentiation in a multi-tenant SaaS application. It involves changes in the core of the SaaS application and its code base in order to comply with tenant-specific requirements that go beyond the configurable limit. Customization incurs an additional layer of application engineering complexity and additional maintenance overhead, making it a much more costly approach for SaaS vendors compared to configuration (Walraven et al. 2014). This kind of an extensible application should provide a pre-defined interface to enable adding and integrating new source code (Tsai & Sun 2013). According to Sun et al. (2008), customization is becoming more complex in SaaS context, including downsides like vendors having to maintain all the customization code tenant by tenant, and having to make sure that upgrading the application should not lead into losing of any single tenant's customization code.

Fehling et al. (2014) remark that in this design, the architect always has to consider the tradeoff between IT resource homogenization, which is necessary between tenants, and customizability that has to be taken into account when opting for one of the multi-tenancy patterns. The degree of resource sharing should be maximized in multi-tenant applications, which leads to better resource utilization and reducing the running cost of the application.

3.3 Variability points

Kabbedijk & Jansen (2011) state that in software, the concept of variability was first introduced in the area of software product lines, in which variability is defined as the ability of a software system to be efficiently extended, changed, customized or configured for use in a particular context. Variability can be divided into three levels within multi-tenant SaaS applications: *Low,* consisting of visual representation of the product, *Medium,* consisting of changes in logic-tier, and *High,* in which the variability influences multiple tiers at the same time, including also tenant's ability to run custom code. Variability is exploited in order to maximize the potential customer base of a multi-tenant application by extensively supporting tenants' requirements (Saleh et al. 2014).

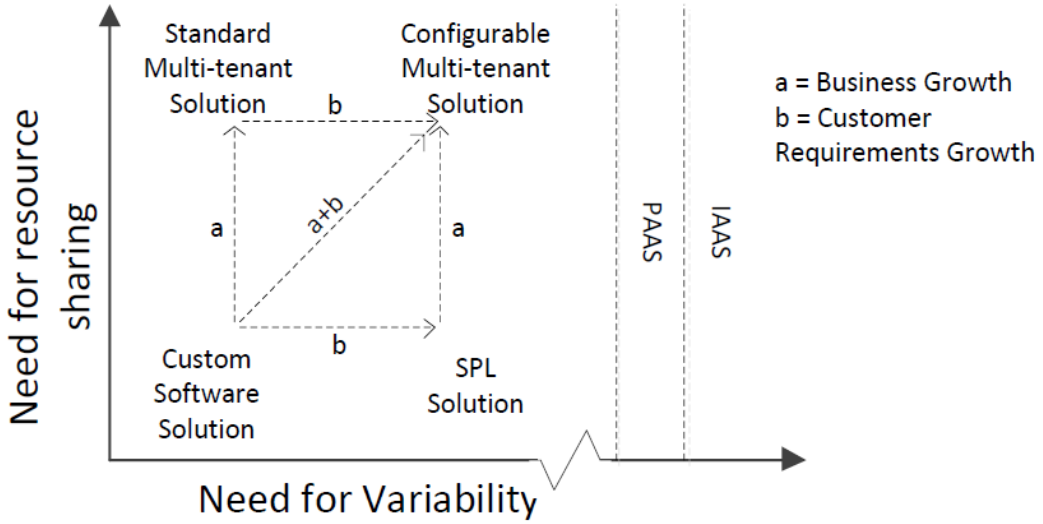

Figure 3.1. Level of variability versus number of users. From (Kabbedijk & Jansen 2011)

The overview of variability and different options available in software deployment are shown in the Figure 3.1: On the horizontal axis is the demand for variability, while the vertical axis depicts the number of customers. A standard *custom software solution* is sufficient for small software vendors having only a handful of customers with specific wishes. As the business grows, the vendor might invest in a *standard multi-tenant* solution because of the advantages in maintenance. *Software Product Line* (SPL) approach suits well to create a variant of an application if the amount of tenant specific requirements increases, but suffers from extra maintenance. Finally, a notable amount of customers with specific needs makes a *configurable multi-tenant solution* the best solution for vendors, benefitting from better performance and maintenance. (Kabbedijk & Jansen 2011)

According to Sun et al. (2008), having a large number of unique clients eventually causes an increase in requirement variance. The requirements for the alterations can be derived from different sources such as regulations, culture, differences in customers' behavior or industry focus, or differences in operation strategy. Hence, the most of the enterprise software applications end up being tailored to serve a specific client. Variability can be divided into two groups based on the origin, *Segment Variability* and *Tenant-oriented Variability*: in the first one, product variability is based on the segment of a tenant is part of, and variability in the second one is based on specific requirements of a tenant Kabbedijk & Jansen (2011). After gathering the requirements, the process usually involves analyzing and classifying tenants' requirements, considering variation points and determining the level and corresponding layer, and selecting appropriate means such as architecture, technique or tools to realize variability in the purposed variation point (Saleh et al. 2014).

Krebs et al. (2012) list three high level design concerns that can be the key differentiators for competitors: customizability, performance-isolation and QoS differentiation. First one is including both configuration and customization concepts, the latter two are design concerns addressing service performance. Performance-isolation expresses how workload of one tenant affects other tenants, and it can be divided into three types: *unisolated*, *weak-isolated* and *performance-isolated*. In unisolated system, tenants that are producing workload within the limits defined in SLA can be affected by high workload caused by others. Weak isolation handles restricting disruptive tenants to some extent. *Performance-isolation* prevents high workload from affecting others, and guarantees that tenant's performance is not hindered by other tenants. QoS differentiation makes it possible to provide services of varying quality depending on the tenant. Quality can consist of giving more resources for better performance, allowing more requests to the system and faster response times. Marinescu (2013) state that performance-isolation is a critical condition for implementing QoS.

Customizability includes modifying assets such as:

- UI: Changing the look and feel of the product can be considered as the most elementary type of personalization. Customizability should include the management of e.g. logo, theme, layout and fonts. (Tsai & Sun 2013)
- Workflow: Workflow is tightly coupled with the business logic. Tenant could compose new own workflow templates using existing types of services or choose an existing template from common repository (Tsai & Sun 2013). Modifying workflow should support switching tasks on and off, adding new ones, reordering tasks, and changing roles that are involved in the process (Sun et al. 2008).
- Reporting: Tenant may request a personalized reporting view to ease the task of analyzing the application data. Creating new, different reports can be accom-

plished by diverse querying, changing data set, and changing the representation between chart, table or graph format. (Sun et al. 2008)

- Data: There can be a need for configuring the data structures: adding and removing data, and modifying existing data by changing field names and types can be enabled in the MTA. It is important to notice that changes to the data may have impacts elsewhere in the application, e.g., the changes should be reflected also on the UI side. The SaaS vendor has to carefully map the relationships between different software artifacts. (Sun et al. 2008)

Kabbedijk et al. (2015) state after having examined the multi-tenancy topic that variability poses an importance also in the future. It is a theme that is seldom mentioned in the literature, but due to its challenges it cannot be ignored in the research.

3.4 Competency model and strategies

SaaS vendors need to take a well-designed strategy to enable self-serve configuration and customization by their customers without changing the base code because of the subscription based model. *Configuration and Customization Competency Model* is something that can be used when launching a new service and considering the degree of supported variance. It is a model that SaaS vendors can benefit from in order to plan and evaluate their capabilities and strategies better. This model separates competency to five different levels based on the supported variability. These levels are *Entry*, *Aware*, *Capable*, *Mature* and *World Class*, ranging from completely standardized offering to a fully tenantized offering for any individual tenant. (Sun et al. 2008)

An application on the Entry level is a highly standardized offering without support for configuration or customization. With no variability points available, the offering needs to have well designed functionalities to cover the needs of the targeted customers. Aware level means that the service has a relatively standardized setting of pre-defined variability points. Variance in this kind of offering is low, but tenants are provided with a parametrized configuration. SaaS vendor can have success on the market even though the competency is on the Entry or Aware levels, but it requires a proper strategy with well-defined customer segments and a deliberate scope of variance. Level of competency being Capable, the service can be regarded as supporting medium level of variance by offering a self-serve configuration tool to empower customers. But even with the user-defined configuration, the service is still relatively standardized. (Sun et al. 2008)

The last two levels, Mature and World Class, are supporting customization of the SaaS service and are approaching the other end of *fully tenantized offering*. The first one means that the service offering has a programmable environment which enables user preferred customization. This approach implies providing a scripting-based platform for flexibly altering the service, thus supporting high variance level. The description for the latter is that the offering includes a platform supported by programming model and tools

which enable very high customization possibilities or even makes possible to create new applications with it. In theory, using these high end competency levels implies that the SaaS service can acquire more paying customers on board because it extensively supports complex variance requirements of potential users. The presented competency model can be used in the assessment of SaaS application to identify possible improvement goals around configuration and customization, while at the same time comparing and evaluating against the market leaders. (Sun et al. 2008)

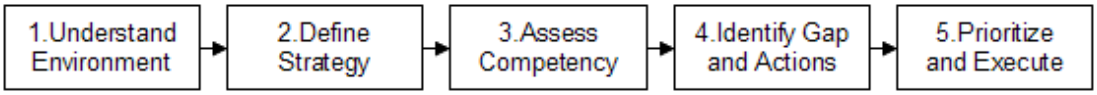

Figure 3.2. A framework to plan and execute configuration strategy. (Sun et al. 2008)

Figure 3.2 depicts the framework introduced by Sun et al. (2008). The framework aims to help SaaS providers to find the right strategy for supporting customizability. Planning and executing a configuration strategy starts with investigating the environment in which the target SaaS operates. First of all, client requirements have to be researched to identify targeted customer segment and the required variance scope. SaaS provider should focus on a segment where customers have similar, relatively low level of variance requirements, and where customers have relatively weak capability to acquire an alternative solution for the purpose. SaaS has to be developed in a way that the application function scope broadens this segment to make the SaaS relevant for as many customers as possible. Secondly, market leader's competency level has to be examined, which may have an impact how the SaaS will implement configuration and customization perspectives to position itself in the market. Other segments may be either too competitive or there are not many potential customers to make the SaaS profitable.

The second step is to define the strategy about how to plan the support for configuration and customization requirements in the preferred segment. There are four models: *Native design*, *Smooth Evolvement*, *Pulse Evolvement*, and *Failure Management*. Native design means that multi-tenancy and differentiation have been taken into account in the design from the beginning to provide high scalability. This approach involves providing sufficient tools for the tenant to handle all the configurations and customizations by themselves, and SaaS vendors abstain from changing source code for any tenant. Smooth Evolvement model requires that SaaS vendors have tools to manage the costs of making changes to application code according to tenant's requirements. Nevertheless, SaaS vendors are required to put effort to support every tenant requirement. In the third model, Pulse Evolvement, SaaS vendors collect requirements from a group of tenants and upgrade the application based on the requirements, trusting that there is potential benefit for implementing the changes. Last model is Failure Management, in which each tenant's requirement causes SaaS vendor to change application code. There are no effective tools and process to manage the cost spent on each tenant, which eventually will cause the failure of the SaaS. (Sun et al. 2008)

Evaluating the competency of the existing SaaS is the third step in the framework. For this purpose, the aforementioned competency models can be applied in more detail to each variance category: data structure and processing, organization structure, user interface, workflow, business rule, and reporting. After analyzing the level of competency for each variance category, next actions could be derived from comparing to the market leader, and finding improvement areas that would give some competitive edge in the market. As the last definite step is to prioritize the actions, and begin the execution of the chosen approach. (Sun et al. 2008)

3.5 Salesforce.com – an example of the market leader

Some applications offered as SaaS can be extended using custom code in which case, the SaaS provider commonly offers a well-integrated PaaS cloud to host custom extensions to the SaaS application. *Salesforce.com* and *Force.com* are good examples of this. Salesforce.com started out as a SaaS provider, but shortly after started its PaaS offering Force.com to haul more customers by providing an elastic platform on which the functionality of the software can be extended and integration to existing applications is made possible. (Fehling et al. 2014)

Salesforce is the world's leading Customer Relationship Management software and enterprise cloud ecosystem. It offers wide range of tools and applications for companies to solve tasks related to e.g. sales, marketing and customer service, helping companies to become more efficient and profitable. Since its initial launch in 1999, it has so far acquired a user base of 150 000 customers, serving big enterprises such as Unilever, Electronic Arts and Spotify, and without forgetting the needs of smaller companies. Salesforce markets itself as the pioneer in the use of cloud computing in enterprise solutions. (Salesforce.com 2017c)

Force.com, being the foundation of Salesforce.com's successful applications, such as Sales Cloud and Service Cloud, is an application development platform which is also provided for individual enterprises and service providers. It houses all types of business applications, including supply chain management, billing, accounting, compliance tracking, human resource management, and claim processing applications. The platform supports more than 100 000 organizations, more than 200 000 deployed applications and millions of users, all of which sets challenging requirements for robustness, reliability and scalability attributes of the platform. In the heart of Force.com is a so-called *metadata-driven software architecture* that makes the platform fast, scalable and secure, and also makes multitenancy possible. (Salesforce.com 2016)

According to Salesforce.com (2017a) Force.com allows tenants to configure and customize the applications by many ways to support the specific business needs the tenant may have. This personalization scope includes page layouts, processes, assignment rules, sharing and security settings. Customization is made possible with VisualForce

pages, which is a component-based user interface framework (Salesforce.com 2014), and with a proprietary language called Apex (Salesforce.com 2017b) which is used to call Force.com API in order to add custom business logic. Salesforce.com (2017a) describes that in order to support such a large user population and vast customization possibilities, Force.com has been built for cloud computing and with multitenancy in the first place in its design. The metadata-driven architecture at its core, Force.com stores everything as *metadata* – literally data about data – in to the database. This means that everything related to tenant's customizations, be it code, configurations or apps, are specified as metadata.

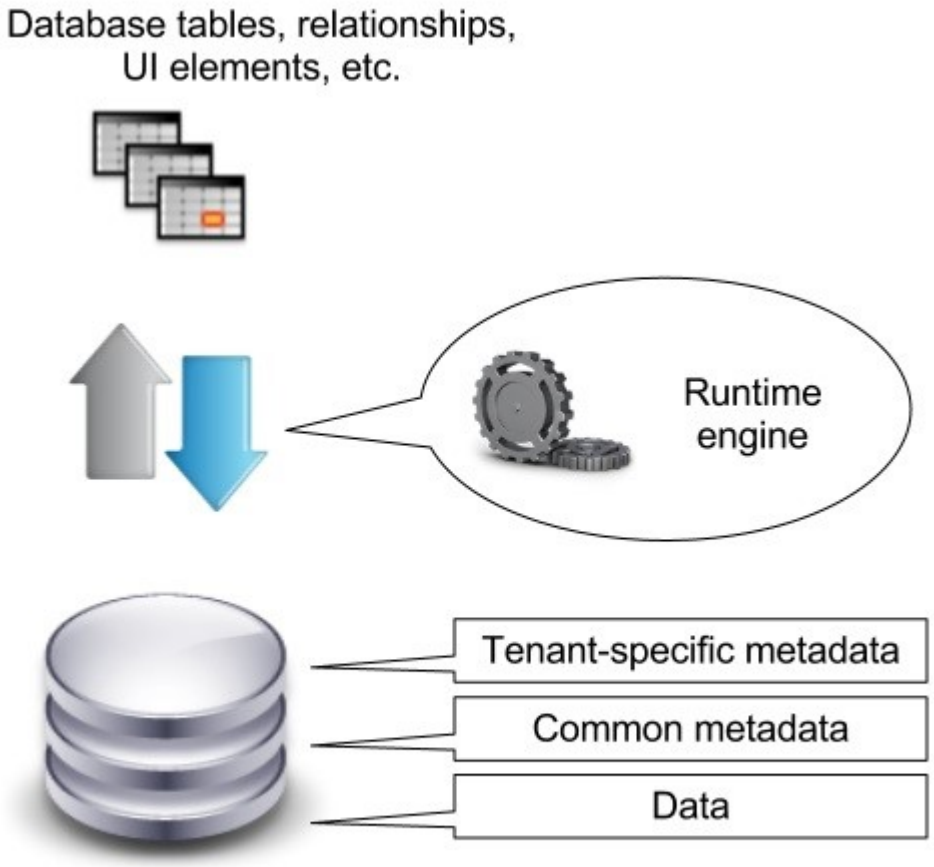

Figure 3.3. Overview of Force.com metadata-driven architecture. (Salesforce.com 2016)

In Figure 3.3 is shown the overview of the Force.com metadata-driven architecture and how Force.com utilizes the underlying metadata with the help of a runtime engine. The process eventually aims to materialize all the application data from the stored metadata. This materialization is done by a runtime engine, also called as *Metadata-Driven Kernel*, which makes the multitenant platform as dynamic as possible and thus fulfilling the individual requirements of various tenants and their users. Without this kind of an engine, it would be difficult to create a statically compiled system executable that would

survive the challenges of multitenant environment. Metadata-driven architecture also takes into account the separation of tenant data, compiled runtime database engine (kernel) and the metadata, aiming to ease the update process of the service. The distinct boundaries make it possible to update the engine and the applications and schemas created by tenants, without risk of one affecting the others. (Salesforce.com 2016)

At the heart of the Force.com persistence solution is a relational database engine with a specialized data model optimized for multitenancy. The shared database includes a centralized repository called Universal Data Dictionary (UDD). This is where each and every logical database object is located; database tables, fields, stored procedures and database triggers are all abstract constructs that are stored only as a metadata. Force.com does not create an actual table in a database or compile any code written by tenants, but instead stores metadata which will be used by the system's engine to generate the virtual application components at runtime. Because of this, modifying tenant's application schema can be done by running a simple non-blocking update to the corresponding metadata. In order to control the excessive access to the metadata and preventing performance issues, Force.com uses immense metadata caches to maintain the most recently used metadata in memory. In addition to the UDD and metadata caching, Force.com's polyglot persistence also includes tables for storing tenants' data (with organization identifier as the key in the row), pivot tables to enhance performance, a full-text search engine and queues for asynchronous background processes. (Salesforce.com 2016)

Multitenant system has requirements for data and performance isolation of tenants. Salesforce.com has tackled these challenges by closely monitoring and analyzing, for example, the code execution, how much CPU and memory resources can be consumed and how many outbound Web service calls can be made. To prevent malicious or unintentional clogging of shared system resources, Force.com's optimizer discards queries that it considers too expensive to execute. This is inevitable in order to maintain the scalability and performance of the system. Moreover, salesforce.com has established a deployment process for applications written by tenant: the process requires, e.g., encompassing unit tests before the code can be approved and certified for production. Later on, the same tests are executed every time Force.com is updated in order to prevent existing tenant customizations from breaking down. (Salesforce.com 2016)

3.6 Multi-tenant application maintenance and challenges

Bezemer & Zaidman (2010) compare the multi-tenant approach to single-tenant environments. In single-tenant setups, every tenant can have his own customized application instance, and this is often done by creating branches in the development tree. In multi-tenant environment this is not possible, and configuration options need to be integrated in the product design: maintenance is therefore more difficult because of the increased code complexity.

At the beginning, a SaaS application is typically developed by focusing on the needs of the first customers to shorten the time-to-market. The initial development and release cycles might not include multiple variability points, but as the SaaS offering becomes more successful, new variations are added to serve new tenants and therefore increasing amount of tenant-specific configurations have to co-exist at run time. This eventually leads to cramming the SaaS offering with substantial amount of small variations, and the implementation might become difficult to handle. (Walraven et al. 2014)

Walraven et al. (2014) express that this scenario has been encountered in practice, caused by the lack of methodical support for the development and customization of multi-tenant applications. The experience from business cases shows that there are two main challenges: how to manage and reuse the different configurations and software variations in an efficient way, and how to implement and support self-service configuration management in tenant-driven customization approach. Both aforementioned challenges affect the scalability of the service. An extra overhead should be avoided provisioning new tenants. Self-service helps realizing the scalability benefits by shifting some of the configuration efforts to the tenant side. The application could allow tenants to manage their tenant-specific requirements by themselves, and the run-time configuration process should be automated. If self-service approach is chosen, the vendor needs to take into account that tenants might require additional support to manage the configurations.

Marinescu (2013) brings up security as one of the main challenges in multi-tenancy by stating that multi-tenancy is usually the root cause of user concerns. Security threats are different between cloud delivery models, for example, SaaS may store users' private information such as name, address, phone number and credit card numbers on one server. These are known risks in multi-tenant systems. Nevertheless, multi-tenancy is one of the key components in utility computing and users have to live with the idea that private data of large number of users may be exposed when the server security is compromised.

Guo et al. (2007) note that differences between each tenant bring new requirements related to upgrading the service as there might be variations in the agreed QoS level. Most of the time all the tenants are not equal, and some organizations relate to service disruptions in different manners, for example how costly such interruptions will eventually end up. Because SaaS service should be able to comply with unique tenant SLAs, service should support customizable schedule of updates to ensure the conformity to the specific agreements. There might be a need to have some monitoring over the used computing resources just to make sure no tenant blocks the resources and renders the application unusable for everyone.

Krebs et al (2012) points out *affinity* as one of the high level design concerns of MTAs. Affinity defines how the requests of different users of one specific tenant are bound to processing nodes. The method is based on tenant specific attributes for the routing of

requests and not user specific ones as in traditional request-response based systems. Distribution of these users becomes an issue when an MTA has hundreds of instances serving tens of thousands of tenants. There are different types of affinity: non-affine, affine, cluster affine and inter-cluster affine. Non-affine type implies that each server receiving a request does not care about which tenant sent the request, thus every request can be handled by any instance. Affine type does not allow distribution of users of specific tenant among several instances. The same application instance handles all requests from one tenant, but it can handle other tenants at the same time. This type is usable if sharing among instances is not feasible, or if the performance is desired to be boosted by efficient, centralized caching for specific tenants. Cluster affine type uses subgroups consisting of multiple instances, and the subgroups serve multiple tenants. One application instance belongs to only one subgroup. In Inter-cluster affine type an application instance can be part of multiple subgroups, e.g., one server in Germany is also located in the EU.

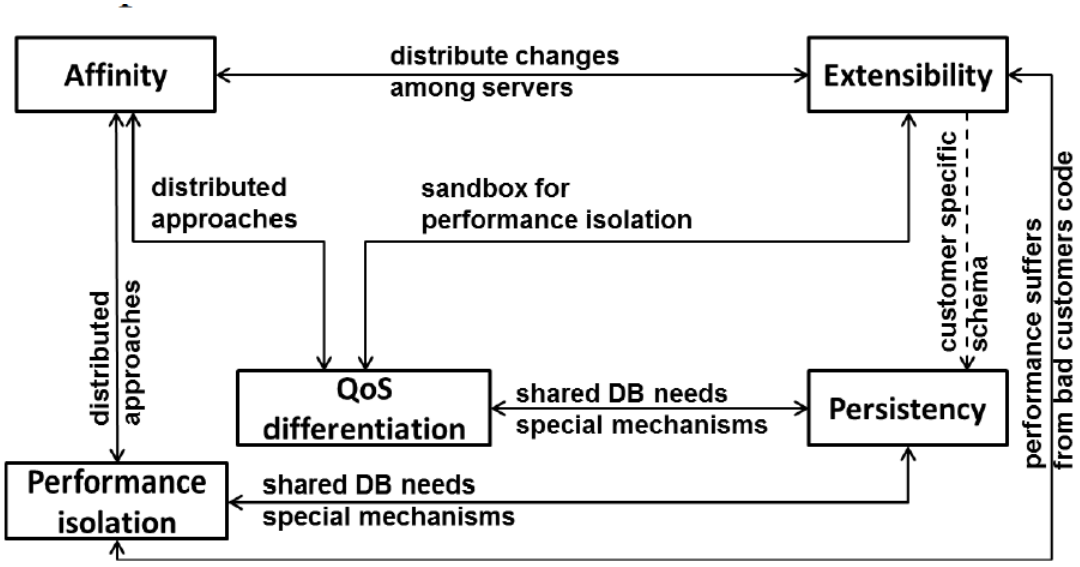

Figure 3.4. Interdepencies between different architectural concerns. (Krebs et al. 2012)

Based on the work by Krebs et al. (2012), Figure 3.4 shows how different architectural concerns influence each other. Affinity choices are affected by the decisions made related to extensibility, performance-isolation and QoS differentiation, because all three require a centralized mechanism to ensure their goals. For example, implementing customizability includes having a mechanism for deploying changes to all instances. Extensibility also requires a sandbox approach to isolate the execution of tenant code so that the whole system does not suffer from poorly written code, which on the other hand affects both QoS and performance-isolation. From persistency point of view, having a shared database may become a major challenge if no isolation techniques are applied to the database: one single query on the database might result in notable performance issues for every tenant. Different storage capacities could be used to provide QoS differ-

entiation, and database schemas could be altered for tenants to support modifications and code extensions.

4. PATTERNS AND DESIGN PRINCIPLES FOR A MULTI-TENANT CLOUD

Ochei et al. (2015) describe that serving a customizable application like an MTA is a complex task. In cloud environments, multi-tenancy introduces significant challenges when deploying and serving application components with different degrees of isolation between tenants. Isolation requirements may be defined by corporate regulations or by law, or by simply declaring a component to be too critical for sharing. On the other hand, a component can be also entirely shared which results in higher utilization of resources. Thus, this leads to a problem where developers have to consider the trade-offs between performance, system resources and access rights when selecting an approach for a multi-tenant application. According to Chong & Carraro (2006), developers also have to remember that any extension of tenant origin requires a corresponding extension to the business logic and to the presentation logic. The former is needed in order to utilize the custom data, while the latter allows users to supply and receive the custom data.

This chapter concerns development patterns and methodologies that can benefit a multi-tenant SaaS application and its implementation. Presented material includes techniques and models that are used to create customizable applications in different kinds of environments in order to give better understanding of the scale of multi-tenancy. Section 4.1 describes the main concepts behind cloud application development by introducing cloud-native applications. Section 4.2 presents tenant resolver and tenant context which are essential parts of multi-tenant data isolation, Middlewares and frameworks are introduced in Section 4.3. Last Sections follow the patterns introduced in the book “Cloud patterns” by Fehling et al. (2014) which are used to provide different degrees of isolation: Section 4.4 presents shared component, 4.5 presents tenant-isolated component and 4.6 is devoted to dedicated component.

4.1 Building cloud-native services

Fehling et al. (2014) describe the term *cloud-native application* as application that complies to the essential cloud characteristics: access via network, on-demand self-service, pay-per-use, resource pooling and rapid elasticity. Handling of varying workload is an important aspect in these kinds of applications. Being able to elastically scale the service is a requisite if pay-per-use and rapid elasticity is going to be used. Cloud-native application can be built on two fundamental principles called *distributed application* and *loose coupling*.

Developers and architects should be aware that a cloud-native application is almost always a distributed application in which the application functionality is decomposed to

independent components that provide a certain function. Alongside with the distributed application pattern, system's elasticity is strongly tied with loose coupling pattern. Dependencies between components are kept minimal in order to simplify scalability, failure handling and update management. One option in loose coupling pattern is to use a message queue, which enables asynchronous communication for the distributed application components. Such intermediary requires that the exchanged data is in a supported format in both ends so that it manages message addressing and routing. Loose coupling might add extra overhead as the communication data has to be serialized and deserialized, thus it may have an impact on the application performance. Loose coupling promotes using data access components to ensure flexible communication between cloud storages and rest of the application. (Fehling et al. 2014)

APIs are critical components in cloud applications, distributed applications and loosely coupled components. For example, in cloud service models, SaaS makes it possible to integrate to other services and applications with APIs, PaaS enables adding plugins and extensions, and instances can be started and stopped in IaaS. Other purposes include e.g. authentication like OAuth, or fetching and modifying resources like in the client-server type web API called Representational State Transfer (REST) API. REST has been one of the corner stones in web services, but recently more lightweight alternatives have emerged such as JSON-pure API (Mikowski 2015) and GraphQL (GraphQL 2017). According to Mikowski (2015), JSON-Pure API's purpose is to provide dynamic messages instead of static content. For example, when image data is delivered via API, the message includes only a URL to a cacheable image. The interface relies only on POST method which includes the possible data part and one of the typical action types: create, retrieve, update, delete or flush. The flush action is a rather unusual type, but it can be used to signal the server that the client has removed data from its storage. Johanan (2014) explains that one of the main perks is the agility and increased development speed, because developers do not need to be aware of the underlying implementation or complexity when using these APIs. In addition, usually application's security is increased when using robust, well-known interfaces.

Vertical and horizontal scaling was earlier mentioned in the Section 2.3 page 9 about virtualization and VMs, but the same principles apply to systems with web servers. When the system's throughput is at risk, horizontal scaling can be used to add new instances or servers to the system in order to cope with the growing workload. Handling the traffic and sending the requests to one of the servers is done with a separate *load balancer* server. Application state cannot be stored locally on the server, because it is likely that the responding server will change in between requests. For this reason, there should be a centralized storage that saves the temporary application state and makes the state accessible for multiple processes on multiple machines. However, if there is a need to store data for a longer time, then a database is a better option. Another, yet important, storage is *cache*. Requests and their responses are stored to cache for later usage so that

servers do not have to spend valuable time and resources in processing the same requests again and again. Developers benefit from PaaS and IaaS service models which usually have means for setting up more instances easily, and which also enable vertical scaling by giving the cloud consumers the possibility to tune the server's computing resources. (Johanan 2014)

4.2 Tenant context, tenant resolver and tenant data isolation

According to Fehling et al. (2014), the first thing to do before any customization can be done is to define the means for identifying the user and the group he or she belongs to. When application components are accessed by authenticated users, the requests have to be embedded with a *tenant identifier*. In other words, components are accessed under *tenant context*. Sharda et al. (2014) further explain the role of tenant context: the encapsulated information in the request may also contain configuration or system information or references to services, which are passed to other objects during processing of a service request. A context object can be valid for a single request or for a single user session. It is advisable to use a context manager for this sole purpose of creating, serving and destroying context objects.

The application stays generic until the user has been resolved. A *tenant resolver* component can be used to find out the tenant identifier from the request. There are multiple ways of handling the resolution, for example, it can be based on URL, query parameters, host headers or authentication data. (Sharda et al. 2014; Walraven et al. 2011)

The last needed piece in the MTA to enable tenant data isolation and customizations is a *multi-tenant data storage*, which is used to store tenant-specific configurations and customizations. It is important to notice that the application shall be prepared to provide default configurations and feature implementations, if the multi-tenant data storage API does not return anything for the requested tenant. Thus, it is necessary for the SaaS provider to specify a configuration which maps each feature to a default implementation. For optimizing the system, a cache should also be taken into use with tenant's customizations. Configurations can be cached or even a whole component injected with tenant specific implementation of a variability point could be stored to cache, so that requests associated with the same tenant identifier and variability point could be handled directly from the cache without putting a load on the databases. (Walraven et al. 2011)

SaaS vendors have to solve how to separate tenants' data in the multi-tenant data storage. This is also one aspect of resource sharing. Storing the underlying data and choosing a multitenancy isolation approach depends on the data characteristics: physical characteristics, performance requirements, volatility, volume, regulatory requirements, transaction boundaries and retention period (Kavis 2014). According to Chong et al. (2006), the architecture has to be robust and secure in order to convince the customers who are “concerned about surrendering control of vital business data to a third party”.

There are three ways how to design the data structure: *separate database*, *shared database with separate schemas,* and *shared database with shared schema*. The decision depends on the amount of tenants and their users, concerning regulatory and cost considerations. Separate database provides the highest isolation level, makes per-tenant extensions easy to implement, but on the other hand operating costs are significant. Also, this approach is probably the best choice if expected amount of users or stored data are high. Using shared database with tenant-specific schemas has the same perks as a dedicated database, and it allows having more tenants on the same server. As a downside, data restoration may be harder and a more time-consuming task. Completely shared database with shared tables is suggested to be chosen if the application is meant for serving great amount of tenants. Having all the data in the same database can provide the same strong data safety as separated ones, but it means using more sophisticated design patterns to ensure the security. Shared approach tends to have higher initial costs caused by development complexity but eventually the operational costs will decrease as more and more tenants utilize the system. (Chong et al. 2006)

4.3 Middleware layer and frameworks for true multi-tenancy

One way of solving the complex task of making a customizable application like an MTA is to use middleware. Sharing a middleware and provisioning a separate application instance for each tenant has earlier been used as a way to provide multi-tenancy. But middleware can be also used in true multi-tenancy to serve application components for each tenant instead of application instances, leveraging economies of scale (Walraven et al. 2011; Gey et al. 2015). Lee & Choi (2012) note that previous works on framework tend to be too complex and are not general enough to be widely used, for example, many of these SaaS solutions using middleware are based on *Service-Oriented Architecture* (SOA) (Walraven et al. 2011; Gey et al. 2015; Guo et al. 2007), but nevertheless they provide a good viewpoint of the subject and how non-trivial issues of tenant customizations can be approached; there seems to be no definitive answer for how to implement multi-tenancy in web application environments because of a plethora of available technologies.

The middleware introduced by Walraven et al. (2011) targets multi-tier, enterprise applications, and proves that creating multi-tenant application with flexibility to adapt to varying tenant requirements can be done with preserving the operational cost benefits from multi-tenancy. The proposed middleware layer has a tenant resolver, tenant context and multi-tenant data storage. The layer uses a *feature-based approach* with predeclared variability points for multi-tenant software variations, which can be modified via an API by the SaaS provider. For tenants there is a configuration interface for activating preferred feature implementation. Based on the configuration files, middleware's tenant-aware module decides at run-time for each variation point in the application which implementation needs to be used. These variation points need to be tagged by

developers to denote the possibility for tenant-specific variations. One possibility to activate the customized feature is to use *dependency injection* pattern, for which there are frameworks available, such as Guice depency injection framework that is supported by some PaaS solutions. Other notable alternative to provide tenant-specific injections is aspect-oriented software development. Good to note that the chosen pattern may be affected by the environment and especially the programming language: most of the enterprise multi-tier applications are written with statically typed languages such as Java and C#, while some multi-tenant SaaS applications utilize dynamic interpreted languages.

There are multiple patterns that support tenant-specific customization through configuration. Sharda et al. (2014) addresses that because the metadata-driven architecture requires careful design and at the same time it drastically increases application complexity, it cannot be considered as the first choice in all cases. The generic logic for runtime execution based on the metadata and the metadata management can be substituted by simpler approaches in SaaS applications that require only minor customizations for a subset of objects. For example, the architect could take advantage of *dependency injection*, *Aspect-Oriented Programming* (AoP), *generative programming* and *GUI templates*.

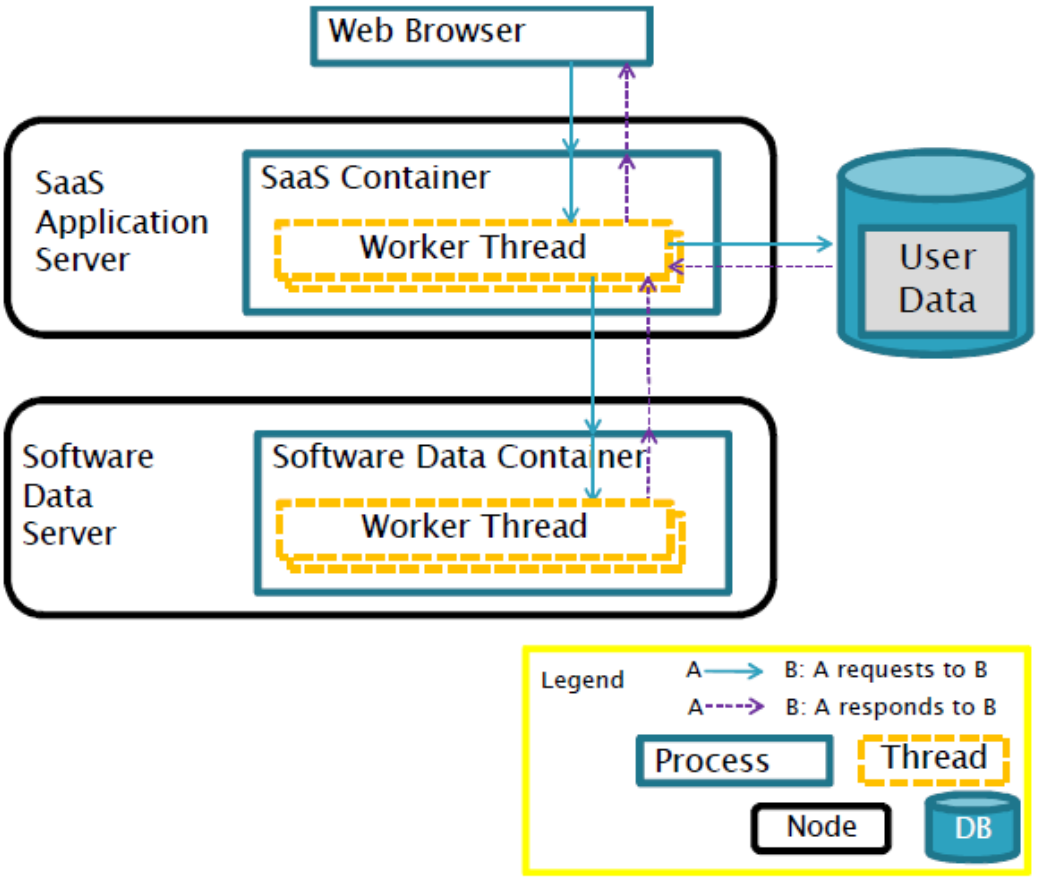

Figure 4.1. Architecture of multi-tenant web application framework. (Lee & Choi 2012)

In Figure 4.1 is shown minimalistic web application architecture for SaaS that supports customization, scalability and sharing of computing resources. The framework does not promote any specific client-side technologies, instead it aims to be a general-purpose, easily applicable without pre-defined boundaries. The architecture is splitted into three main components: a SaaS application server that serves requests from tenants, software data server which is the container for UI and business logic codes, and a database that stores all the user data. SaaS application server handles the request and fetches tenant's user interface and business logic codes, which are dynamically executed for the response message. Database for user data may be accessed when finalizing the request. Scalability is supported by allowing more SaaS application servers to the front. Framework promotes of caching software data in the SaaS application server to mitigate the impact on the performance. (Lee & Choi 2012)

A notable downside of the framework presented by Lee & Choi (2012) is that it does not clearly show the necessity for tenant resolving and request filtering, which may already require a database connection when processing the initial request to the SaaS application server. Another proposal for a framework which also aims to more thoroughly support the whole lifecycle – development and maintenance – of an SaaS application is

presented by Guo et al. (2007). One of the interesting ideas in the provided framework is to make a distinct separation between application components that can be handled any developer and components that require extra care from multi-tenant-aware developers. This simplifies overall development effort and results in an application that is both faster to implement and easier to maintain. Complicated multi-tenancy parts are secluded, which ends in "simulating a virtualized single-tenant application development environment" for most of the developers.

4.4 Shared Component

Fehling et al. (2014) introduce *Shared component* pattern as one of the multi-tenancy patterns that focuses on maximized resource sharing on an application instance and still, at the same time, allowing individual configuration. The provisioning of application components requires optimization by "limiting the portion of the application stack and the number of application components deployed exclusively for one tenant". This can be used if the component only provides data, does not store data of tenants, and all tenants can be treated as a uniform user group who will be granted a common user experience and service level.

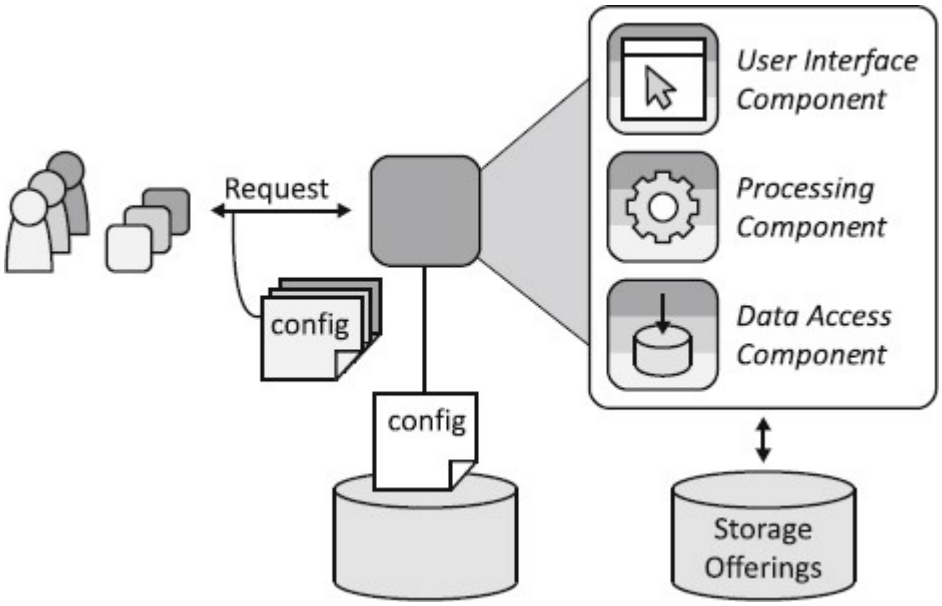

Figure 4.2. Shared component. From (Fehling et al. 2014)

Figure 4.2 depicts the usage of a shared component. Shared component instance handles requests of all tenants, and the components are configured equally for each tenant. Thus, each component instance, in this case *User Interface Component*¸ *Processing Component,* or *Data Access Component*, behaves the same way for each tenant. Storage offering is a component for tenant's data, for example, the storage can be a relational database or a key-value storage. However, the component's configurations allow only minor differentiation between tenants, for example display resolution could be passed to the

component in each request. In addition, the component might not be aware that it is handling workload of different tenants. Downside of this pattern is that a high workload generated by an individual tenant might have impact also on the rest of the users, if the application is not developed to scale well. (Fehling et al. 2014)

4.5 Tenant-isolated component

MTAs aim to have a high resource sharing rate, but this is clearly not always possible. Sharing of application components is hindered by three factors: tenants' unique requirements, security and performance-isolation. Tenants will have requirements for the cloud application and expect it to be configurable to their individual needs. For example, user-interface related properties such as specifying language, color schema and date format are subject to change. Security aspect is relevant when tenants demand strong access control to their application. The third point means that the workload of one tenant shall not affect the others using the same application. Pattern to support this kind of much wider tenant-specific configuration is called *tenant-isolated component* pattern. With it application components can be implemented to be tenant-aware, adding functionality to separate tenants and assuring isolation of access, performance and data storage. (Fehling et al. 2014)

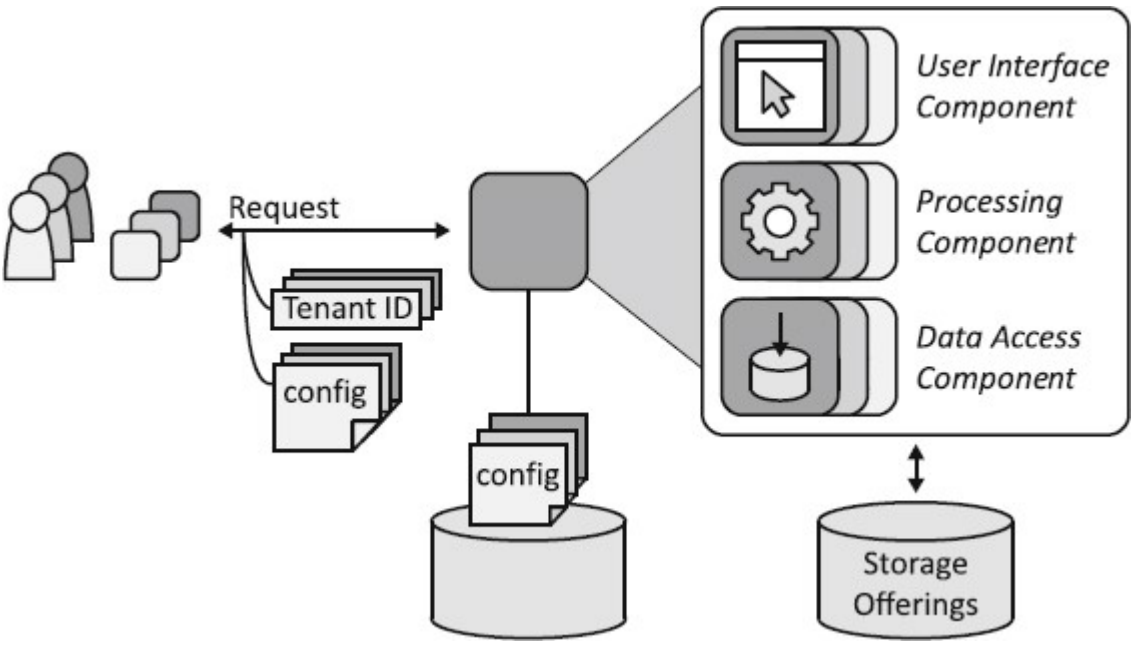

Figure 4.3. Tenant-isolated component. From (Fehling et al. 2014)

Tenant-isolated component pattern is shown in figure 4.3. Every tenant has to be authenticated in order to initialize the accessed component by previously done configurations, for example a tenant resolver component can be used for this purpose. Each request is associated with tenant identifier and component behavior is adjusted. The com-

ponent, be it for user interface, processing, or data access, ensures proper isolation throughout the process by separating the requests with the given tenant identifier. Tenant's configurations can be passed to the tenant-isolated component with every access request, if the size is small enough to make it possible. (Fehling et al. 2014)

This pattern enables the highest degree of resource sharing application functionality between tenants. SaaS provider benefits from lower running costs when tenant-isolation application components are taken into use as the runtime cost per tenant is reduced and utilization of underlying IT infrastructure is increased. If the component is scaled out, the number of instances may respect the workload of all tenants sharing the component. The achieved cost savings enabled by efficient resource sharing may allow cloud application providers to strive for larger market. (Fehling et al. 2014)

4.6 Dedicated component

In terms of customization, there are cases where tenants shall be exclusively handed a component that provides critical functionality for a specific tenant while still allowing other components to be shared between all the tenants. Fehling et al. (2014) introduce a pattern called *Dedicated component*. It can also be beneficial in cases where components should be otherwise configured very specifically for individual tenants. Dedicated component pattern enables tenants to modify parts of the application very flexibly to their requirements.

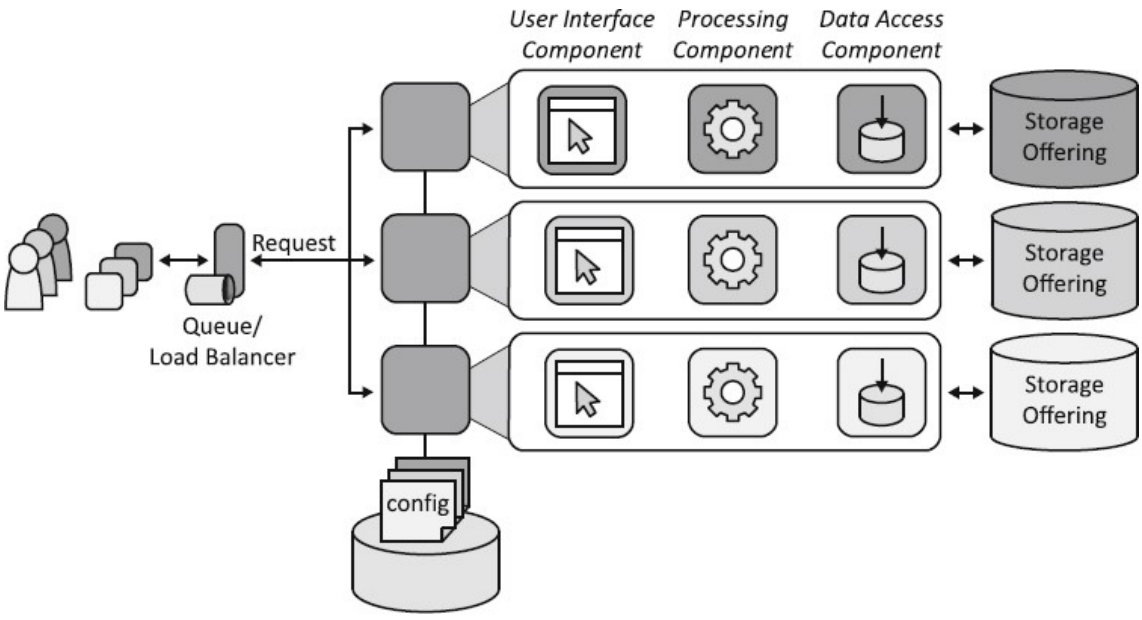

Figure 4.4. Dedicated component. (Fehling et al. 2014)

Figure 4.4 illustrates tenants accessing dedicated components. The application processes a request containing a tenant identifier and routes it to a different application component instances. Each tenant can be exclusively provided a component that has been developed according to the individual tenant's requirements. Dedicated component fits well

with cases where existing applications need to be integrated to a cloud application. A well-known example of Salesforce.com started offering an elastic platform, PaaS, on which custom-developed dedicated components can be hosted. The customer may then use a tenant-isolated SaaS application to handle most of the needed application functionality, while having dedicated components handling the custom functionality. (Fehling et al. 2014)

Due to the nature of dedicated component and its isolation properties, this pattern hinders sharing of application functionality between tenants and therefore restricting resource sharing to lower layers, such as platform or virtualization layers. Previously mentioned shared component and tenant-isolated component patterns should be preferred whenever possible to maximize the degree of resource sharing as the ability of the provider to benefit from economies of scale is reduced by using dedicated components. Using these two aforementioned component models can also be used instead of dedicated component, if only laws and regulations restrict resource sharing. In this scenario, tenant can be given an own component instance which is implemented based on the shared component and tenant-isolated component patterns. (Fehling et al. 2014)

5. CLOUDIFICATION PROJECT AND MTA REQUIREMENTS

Avaintec Oy is a company that has been in the business for 20 years. In this time, the company has grown to be the market leader of digital signatures in Finland and has seen the benefits of taking agile development methods into use as one of the first companies in Finland (Avaintec 2017). As a holder of quality management and information security management system certificates ISO/IEC 27001 and ISO/IEC 9001 respectively, Avaintec strives for secure high-quality products which include also products for document archiving and forms. Recently, the company has gone through a branding face-lift and is on the way of making itself more renowned in the world with its startup mindset. This has resulted in a new cloud project called SignHero, which is an easy-to-use digital signature service for companies.

Section 5.1 introduces the SignHero product family, giving some background information about the target environment and the essential concepts which concerns this thesis. Section 5.2 discusses the requirements for the configuration management tool which allows changing some parts of the service to match better the needs of the customers.

The latter part introduces the most important tools, technologies and environments that have been earlier decided to be taken into use in this project. Section 5.3 presents Node.js which will be used as the web server. Section 5.4 introduces front-end technology called Dojo toolkit which is widely used in the company. The database solutions, Cassandra and Usergrid, are explained in Section 5.5. In Section 5.6 is presented the development environment where the prototype is created.

5.1 Cloudification project and SignHero product family

The cloud project is the next big step for Avaintec to take. It will be a remarkable change considering the new business model, organization and development operations (devOps) because most of the Avaintec products have so far been installed in customer environments. On the other hand, the change could have been even bigger if the previous products were not browser-based applications. In the beginning of writing this thesis, the project was in a really early phase and not all architectural or technical concerns had been solved. But nevertheless, the countdown was started for the release of new SignHero products: *Tailored*, *Kit* and *Express*.

The roots of SignHero product family are in Avaintec's earlier products called X-Web Form Manager (XWFM) and X-Digital Signature Suite (XDSS). XWFM, which inspired SignHero Tailored, is an on-premise service for creating, using and archiving

electronic browser-based forms. Years have certainly shown that the customers have their unique requirements defined by the environment and the business type, and the software has been customized for each tenant case-by-case. XWFM can be considered as a traditional VM-per-tenant application, but the latest version has its own code editor and management tool for handling all the configurations and process flows; this makes it a highly customizable single-tenant application as the customer can handle differentiations by himself. SignHero Tailored is used for custom projects which greatly exceed the limitations of light-weight SignHero Express. XDSS is used to digitally sign data and it stands out as the main building block of all Avaintec products including the new cloud services. It is used in, for example, signing electronic prescriptions.

This thesis is tightly coupled with SignHero Express. It is a SaaS application designed as a single-page application that provides a document signing service for companies. These companies move a part of their business process to the cloud and get essential documents signed by their customers, partners and as well themselves. The service allows starting a new signing process with multiple signing parties, supports tracking the on-going processes and in the end archives the signed documents securely. Signing can be done with one-time-pass code based authentication or with Finnish bank codes. This does not require having an account to SignHero; invited users can be anyone. It is also good to note that SignHero can be used as a standalone version if the signatures are provided only by subscribed users with individual login credentials. High security is promoted with a comprehensive audit trail. As with any sophisticated SaaS service, there is an API available that can be used by an external system, for example, customer might need to integrate its CRM to a digital signature service in order to sign documents in customer's own environment. Rest of the thesis will refer to SignHero Express simply as SignHero.

An essential part of the service is a *signing flow*. Signing flow is the whole process of a document getting signed. The flow consists of signing parties – the creator of the signing flow, creator and other invited users, or only invited users – and one of the aforementioned authentication methods. Signing flow can be initialized by selecting the signers and the authentication method, uploading one or more PDF documents, and providing necessary information like the email addresses to invite other signers to the SignHero service. The service automatically sends the emails as an invitation after the flow has been initialized. The invited signers then sign in to the service with the selected authentication method, and either signs or declines the process in a page called *signing page*. During the signing process, there are email notifications depending on the state of the process: the service informs the owner of the process when one party has signed or declined the document and when the flow has been completed. The subscribed SignHero user can send email reminders to the other signing parties from the signing flow status page. Signing flow becomes complete after all signatures have been ac-

quired from the signing parties. The signed document is available from the service's document archive for e.g. printing purposes.

5.2 Objectives for the configuration management tool

The goal of the thesis is to implement a prototype of a user-driven configuration management tool for SignHero and modify the application so that it utilizes the inputted configurations. The tool, which will be a part of the SPA, aims to provide necessary functionality to support multi-tenancy in order to improve SignHero's state on the markets and thus getting more customers onboard. Even though this is the ultimate objective, initially multi-tenancy and the customizations were not part of *Minimum Viable Service* (MVS). This has a very predictable reason: to shorten time-to-market in order to fill the spot of a new digital signature service on the markets. However, SignHero has been designed with multi-tenancy in mind from the beginning, containing e.g. proper data isolation between the tenants, but customization which is a vital part of multi-tenant applications is not available yet. This thesis can be viewed as a side project to the main service because of the low priority at the current stage, but implementing the tool has a genuine purpose of providing an outcome that meets the demand for customizations.

The requirements for the configuration management tool come from both the customers and from in-house designers. There is a direct need from customers to personalize SignHero, but there is also a list of desirable features that will possibly make it out to the service at some point. In order to keep the scope of the thesis tolerable, the selected features concern only the signing flow and the appearance of the service. Performance-isolation and QoS variations between tenants are out of scope, but tenant data isolation and scalability should be taken into account in the implementation. These selected features are derived from the fact that customers most often want to change look and feel of a service to match to their organization's visual identity:

- The logo used in the service should be customizable so that the customer can upload the organization’s own logo. Customer's own logo shall be used in the signing page that is intended for invited signers without a SignHero account, and the same logo should be visible in emails that are sent to these invited signers.
- User should be able to save a default signing flow consisting of the selections related to signing parties and authentication method. These preferences are then automatically served as an input to the signing flow initializer, so user can start from the PDF uploading page. This aims to save user's time if the organization's signing process is always the same. The same saving feature should be available also in the signing flow start page.

- The configuration management tool is simple and easy to use. There is no need for a complicated editor or preview modes of e.g. emails with the customized logo.
- Some features will not be implemented in the prototype, but are taken into account when designing the tool. These features include the possibility to change the styles and fonts, and changing some parts like a footer element in emails.

5.3 Node.js

Uses of Javascript have expanded to cover also the backend systems of web applications, which can be considered as a consequence of improvements made to the Javascript engines. Open-source Node.js is one of these solutions that has emerged to be a widely used building block for scalable network applications. It is an asynchronous event-driven Javascript runtime that is included with built-in libraries for networking and other I/O operations. Notably, Node.js is powered by Google V8 which can also be found from web browsers Google Chrome and Chromium. Scalability trait comes from the ability to handle many connections concurrently, as Node.js is designed with event-driven model that revolves on events and callbacks rather than threads commonly found in operating systems. This frees software developers from dealing with typical issues related to concurrency such as dead-locking. One of the main reasons why concurrency is supported so well is because Node.js operates on a single thread. Despite this, it also allows multiple processes to be created to take advantage of multiple cores in the environment. (Node.js 2017)

Node.js comes with a package manager full of open-source libraries called *npm* for installing, sharing and distributing code. It makes managing dependencies easy, and benefits developers by allowing to browse the selection of packages for building an application faster. Originally npm started as a package manager solely for Node.js, hence its name *Node Package Manager*, but now it serves all Javascript developers. (npm 2017)

5.4 Dojo Toolkit

Another open-source Javascript library relevant to SignHero.io and this thesis' configuration tool is Dojo Toolkit. Dojo Toolkit can be expressed as a set of tools that is used to develop complete Web applications. Being designed as a “toolkit”, it does not tell how applications should be assembled out of the given components, and does not set constraints on the project size where it can be used. The developer is given free hands to pick whichever the design pattern for the implementation. Dojo's components are divided into four main packages:

- dojo: main part of Dojo containing the “core” functionality. Covers wide range of functionality such as promises, AJAX, DOM manipulation and internationalization libraries.
- dijit: contains a set of widgets for creating the UI. Dojo supports using widgets in both “programmatic style” and in its own “declarative syntax”. Programmatic style means that plain Javascript is used to instantiate the objects, and all the widget's features are expressed in Javascript. Declarative style involves a parser which reads the DOM and creates the widgets which have been decorated with special data-dojo-type and data-dojo-props attributes.
- dojox: an extra collection of packages and modules built on dojo and dijit packages which provide additional functionality and eases development. Modules and packages are not mature enough to be placed in either dojo or dijit packages.
- util: consists of various utility modules for building, testing and optimization. Modules in util package are not meant to be accessed directly from web pages.

Since version 1.7, Dojo Toolkit has been using *Asynchronous Module Definition* (AMD). Modules written in AMD format brings forth many benefits: it allows completely modular web application development, fully asynchronous operation, better dependency management and true package portability. Even though AMD is mostly used in the client-side, AMD loader works well under Node.js. With so-called “bootstrapping”, AMD environment can be taken into use with minimal overhead and without disadvantages. Developer can benefit from code style consistency when using the same Dojo coding style for both client and server, which results in faster implementation of applications. Currently, the newest version of Dojo Toolkit is 1.12 while version 2.0 is under development. Version 2.0 will be written in Microsoft's TypeScript, and in addition to browser environments, the focus will be in supporting modern environments better, which includes Node.js, io.js, and even VR headsets. (Dojo 2017)

5.5 Cassandra and Usergrid

Cassandra (2017) describes Apache Cassandra is a NoSQL database that is suited for large scale applications required to store massive amounts of data, without compromising high availability and performance. It is designed with no single point of failure, replicating all the data across nodes in the cluster. Scalability is achieved by supporting tens of thousands of nodes. According to Saxena et al. (2015), Cassandra is a “hybrid between a key-value and a column-oriented database”, and it is not optimized for storing large files (Cassandra 2017). Alternatives to Cassandra's NoSQL database are storage offerings such as Block Storage, Blob Storage, and Relational Database.

Another important part in the data storage solution is Apache Usergrid which is meant for highly scalable applications. It is a backend framework that consists of Cassandra, application layer and client tier, and contains fundamental web application services such

as user registration and management. Usergrid's data model consists of *collections*, *entities* and *properties*. Essentially all data is stored as entities which belong to a single collection. Entity itself is a JSON-formatted data object which consists of a set of default and custom properties. This is one of the things what makes Usergrid powerful. It supports saving data expressed in JSON format, and the data model does not have a predefined schema that has to be followed, but instead entities and their properties can be flexibly altered. (Usergrid 2017)

5.6 Development environment

The configuration management tool will be mainly developed on a local machine with a dedicated virtual machine. There are some characteristics that are different between the development and production environments, most notably cloud characteristics are absent in the development environment and the production environment uses lightweight Docker containers instead of virtual machine instances. This denotes that scalability is out of scope for the configuration management tool. Otherwise the development environment has the same previously listed tools: Dojo Toolkit, Node.js and Usergrid framework with Cassandra.

The code is stored on a Bitbucket server (Bitbucket 2017), which is Atlassian's on-premise Git version control system aimed for enterprises. Later while there is significant progress in the configuration management tool, an external test server is taken into use where testing shall take place. The code repository is tied with an automated build and deploy tool in a similar manner as in continuous integration.

6. IMPLEMENTATION OF THE MANAGEMENT TOOL

This chapter presents the developed prototype of the configuration management tool. The application is gone through by explaining the decisions related to design and the selected tools, eventually describing the implementation in more detail. Before any implementation work, the current status of SignHero is briefly gone through to verify how the current architecture of the main application is affecting the architecture of the configuration tool that is being done for this thesis. This verification is done in the first Section 6.1. Section 6.2 presents the taken approach and goes through the main design points that are the outcome of the findings from previous Section 6.1 and this thesis' theory chapters. Section 6.3 undergoes the backend solution in more detail and also presents the architecture. Front-end is presented in Section 6.4. Evaluation of the tool is wrapped up in Section 6.5. Future work and improvement possibilities are described in Section 6.6.

6.1 Revising the current architecture

As time of writing this thesis and starting the implementation of the configuration management tool, the cloudification project has been going on for a while. Since the beginning, there has been some natural evolution in the code related to both the backend and the frontend. However, there is also a risk that the architectural design will change over the course of implementing the configuration management tool and enabling customizations in the main service, because the MVS is still being worked on. This uncertainty is considered as a sufficient reason for not making a complete evaluating of the current architecture, but instead a brief revision is done to detect important details that might affect developing the configuration tool. Moreover, it also needs to be checked if there are any changes to be done to the current architecture to support multi-tenancy.

Signhero is initially designed as a multi-tenant system, but the lack of customization possibilities make it closer to a multi-user system. Customization features were indeed brought up in the early steps, but kept separate because of the goal of producing MVS for minimizing the time-to-market. The key characteristics found can be listed as following after reading the source code and the documentation and testing the application:

- There is no middleware such as Express.js for Node.js. For example, with a use of a middleware an extra layer could be responsible of identifying the tenant from the request, and saving the identifier for later use in unified manners.
- The request object or the tenant identifier is not always available during each state of the process. This complicates fetching configurations for tenants.

- There are separate data storages for non-object assets, such as images, PDFs, and object assets. In the current persistence solution, non-object assets are saved to a filesystem storage and automated scripts handle the replication between the nodes. This is subject to change later, but for time being the sufficient approach for this thesis is to use the local filesystem of the server. Without the replication scripts, this approach would make scalability impossible. Object assets are stored to Cassandra through Usergrid features.
- Company of the user can be identified by one of the Usergrid properties. This company identifier is already chosen as the tenant identifier in the service. This makes resolving tenant easier as there is no need to define new means for identifying users.
- UI and email templates are done with a multi-user approach. This means that there is some personalization available, for example, email subject and content can be defined by the user. These are one-time only type of personalization and are not saved anywhere.
- No easy way to get and set the signing flow selections of signing parties and authentication method. This needs to be improved at the same time when the service is extended by adding a save button to the page.

6.2 Selecting the approach

Considering the given requirements, the application to be developed falls into the category of "Entry" level in terms of multi-tenancy. As stated before, this does not make the cloud inferior to other systems because it can still offer enough features for the customer. Inspecting SignHero from strategy point of view, the strategy model does not exactly fit into the classification of "Native Design", even though multi-tenancy has been in the taken into account in the design and there is a way to separate the tenants. Currently there are no effective tools to manage customizations in the main service, for which the strategy model could be defined as "Pulse Evolvement". The SaaS will be improved when worthwhile feature requests appear either from customers or from the in-house development team.

As the current scope of the new cloud application does not include many variability points, the chosen tools and patterns have been chosen to be more minimized. However, the management tool and especially the backend should be possible to be continued in that direction where there are lots of customization needs by the customers. The presented design patterns and frameworks were too heavy to be taken into use in a project of this scale, essentially because the scope does not include code extensions. The metadata-driven architecture familiarized with Salesforce, as well as middleware based solutions were superseded by lightweight methods like UI templates and dependency injection. SignHero is a lightweight service in this sense, and keeping things simple also

from customer's perspective preserves the increased operating costs minimal as there is no need to increase customer support.

It is clear that creating a multi-tenant application is far from trivial when needing to take into account also all the security, scalability and performance isolation matters in addition to creating support for tenant's customized code. Fortunately, the focus in this thesis will be in configurations. However, it is important to keep in mind the scalability of the service. The architecture should be planned wisely and it should support future requests: getting new customers on board should not cause extra development when customers do their configurations.

The current architecture supports identifying and filtering tenants. There are, for example, utility modules for checking the session data from Usergrid, which can be utilized for verifying the details of the user related to the sent request. The minor changes needed are related to having the original request object available or alternatively having the tenant identifier available throughout the process. Backend needs to support the extended data model and some current modules such as signing flow page has to be modified with new functions and widgets. Nevertheless, connectivity to the main service does not require too much extra effort.

Well-designed APIs need to be done for implementing the variability points. APIs should provide means for at least updating, retrieving and deleting logos and workflows. Using JSON-pure API was decided to handle configuration parameters, and for uploading files there is an upload URI. The underlying data model needs to be defined. There are at least two approaches to store tenant configurations: saving the variables to a database or creating a structured configuration file that will be saved to a filesystem solution. Because transferring configuration data to and from backend will be done with JSON as the data format, it is convenient to use the same format to save the data with Usergrid's JSON data model. For configuring emails, it is enough to use templates and pass an address to the image as a parameter, instead of saving image directly to the email which would inconveniently increase the size of the email. Tenant-driven approach leads to having non-functional requirements like usability. For this reason, the focus will also be in the UI features to make the usage as easy as possible for the user.

6.3 Backend

The server-side functionality was created with Node.js with the unique Dojo AMD “bootstrapping”. With the previous experience from Dojo Toolkit, this turned out to be a very convenient way to write code to a Node.js server, speeding up the implementation without the need to learn a new syntax. Furthemore, the package manager npm made it easy to get the needed modules for handling connections to the development environment's local filesystem.

Security and data isolation plays a big role in multi-tenant systems. Authentication will be based on browser cookies and the session data handled by Usergrid features. The configuration management tool cannot rely on the sent cookie data as it can be tampered easily. For this reason, the tenant identifier in the request is validated to make sure that the user really belongs to the group he or she claims to be part of. This is one example how multi-tenant data isolation is verified in the configuration tool.

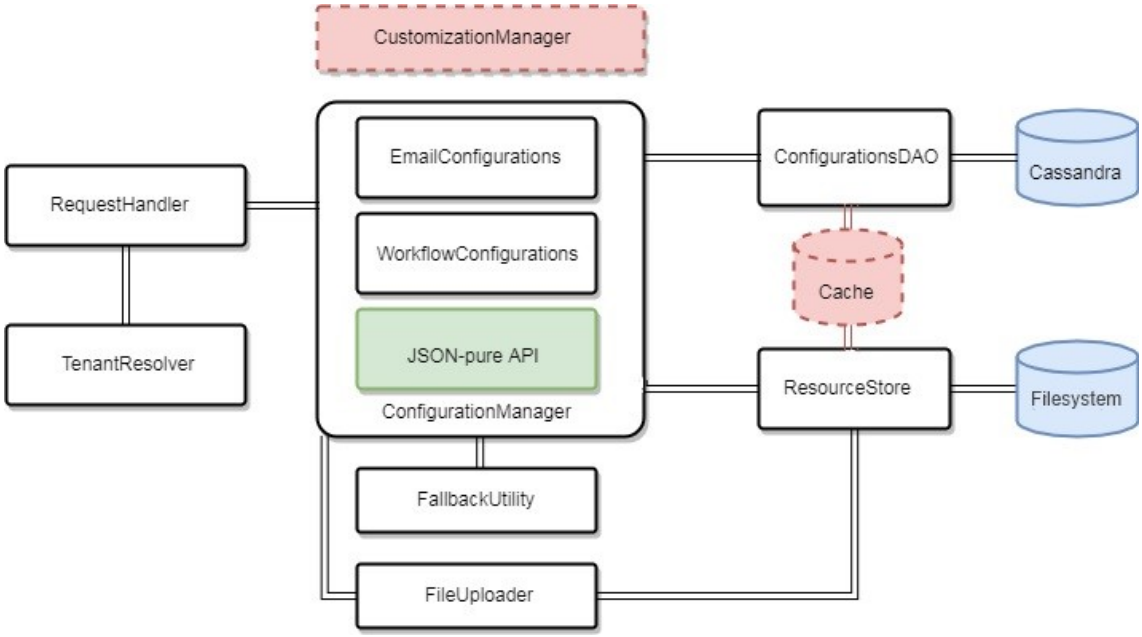

Figure 6.1. Backend architecture of the configuration management tool.

Figure 6.1 depicts the high-level architecture of the implemented configuration tool. In addition, it indicates components in red color that were not created in the scope of the-sis, but which would exist in a sophisticated MTA. For sake of simplicity, possible load balancers and clustered database solutions are excluded from the diagram, as well as relations to main service's utility classes. All the shown modules, except databases marked as blue, are running on the same Node.js server. When a client sends a request to fetch some tenant related data, it first hits the RequestHandler module. As the first vital step after routing the request to the configuration manager, the tenant context and request validity are solved with TenantResolver. The persistence solution is divided into ResourceStore and ConfigurationsDAO depending whether tenant assets or configura-tions are handled.

JSON-pure API accepts requests only from authenticated users. The API currently sup-ports only modifying logo and workflow configurations with *retrieve*, *update* and *delete* operations. The tenant logo, which is shared between signing page and email templates, cannot naturally be behind this kind of authentication because otherwise it would be impossible to render them in the email when using external source in the markup. As an alternative, it would be possible to embed the image directly into the email content, but this would make the message size inconvenient. But on the positive side, it could be

attractive to companies who promote security and do not want to allow public addresses even to an organization logo. Also uploading the file has to be done outside this API.

Implementing the persistence solution required that two things are solved: how to save the configurations to Cassandra and how to save assets such as logos and CSS files to the filesystem storage. All configurations are created as properties inside a new entity called *configuration*, which is linked to a group. Because Cassandra supports JSON-format, it is highly convenient to make any kind of structure, e.g. for workflow was created a property depicted in the code snippet below:

```
default_workflow : {
      flowType: "",
      strongAuthRequired: "",
      inUse:""
      }
```

Albeit the structure is quite minimalistic and it only supports the current version of sign flow parameters, it gets the job done. The property *inUse* is of boolean type and simply notes if the configuration is activated. ResourceStore module communicates with the filesystem storage and creates read and write stream to the tenant specific directories.

After handling storage, the next natural step was to make use of all the stored tenant configurations and implement a hint of multi-tenancy to the back-end part of the main service. Tenant-isolated pattern was chosen as the approach for customized components. All the concerning email templates were updated with a new variable *tenantLogoSource* whose value is eventually injected with a value from database. The main service communicates directly with EmailConfiguration module when the process involves sending emails to the signing parties. Centralized module *FallbackUtility* takes care of fetching default values if configurations do not exist.

6.4 Frontend

The front-end part of the configuration tool was created with Dojo Toolkit and with its versatile set of widgets. The main goal was to create a simple customization page where user can easily update the logo used in the signing page and in the emails, and save the default signing flow so it will not be asked every time user initializes the signing process. Also an option was added to switch back to default choices. Main application has a container for all the panes alongside with a controller that handles the navigation between the panes. Customization pane can be opened by clicking user info on the top right corner. The main service thus handles showing and hiding the pane based on the user actions.

```
signhero-custom
      Configurations
            templates
                  LogoManager.html
                  WorkflowManager.html
            Tools
                  LogoUploader.js
                  WorkflowTool.js
            LogoManager.js
            WorkflowManager.js
      css
            customization.css
      Panes
            templates
                  ConfigurationsPane.html
            ConfigurationsPane.js
      util
            ConfigLib.js
```

Figure 6.2. The application structure of front-end implementation.

The structure of front-end is shown in the Figure 6.2 with slightly modified directory and file names compared to the actual source code. Starting point of the customization pane is a widget called *ConfigurationsPane*. When the SPA is opened, it initializes all the panes that are currently available for the user. At this point in the pane's constructor, Dojo's programmatic syntax is used to create both *LogoManager* and *WorkflowManager* widgets that reside under *Configuration* directory, after which they are placed to their respective positions described by *data-dojo-attach-point* in the ConfigurationsPane HTML-template. With the Dojo's programmatic style it is easier to pass some extra parameters into the widget constructors or apply a check based on the user privileges whether or not the widget is meant for the user. Also changing the content of the customization page is easier, for example, if there is a need to split the features into separate pages. LogoManager widget handles the updates of the organization logo. It has its own template and it contains another widget called *LogoUploader*, which is extended of the basic file loader used in signing flow process. WorkflowManager widget accordingly is responsible of handling sign flow related updates, with identical structure as its logo counterpart. At this project, Dojo's declarative syntax is only used in the templates with the default widgets found in dijit package. Under directory *util* resides one the main blocks of the configuration tool called *ConfigLib*. This is a module created with facade pattern that communicates with the backend API, taking care of resolving tenant based on the browser session and thus simplifying the usage of the module. One master CSS file for customization pane has also been added to the structure.

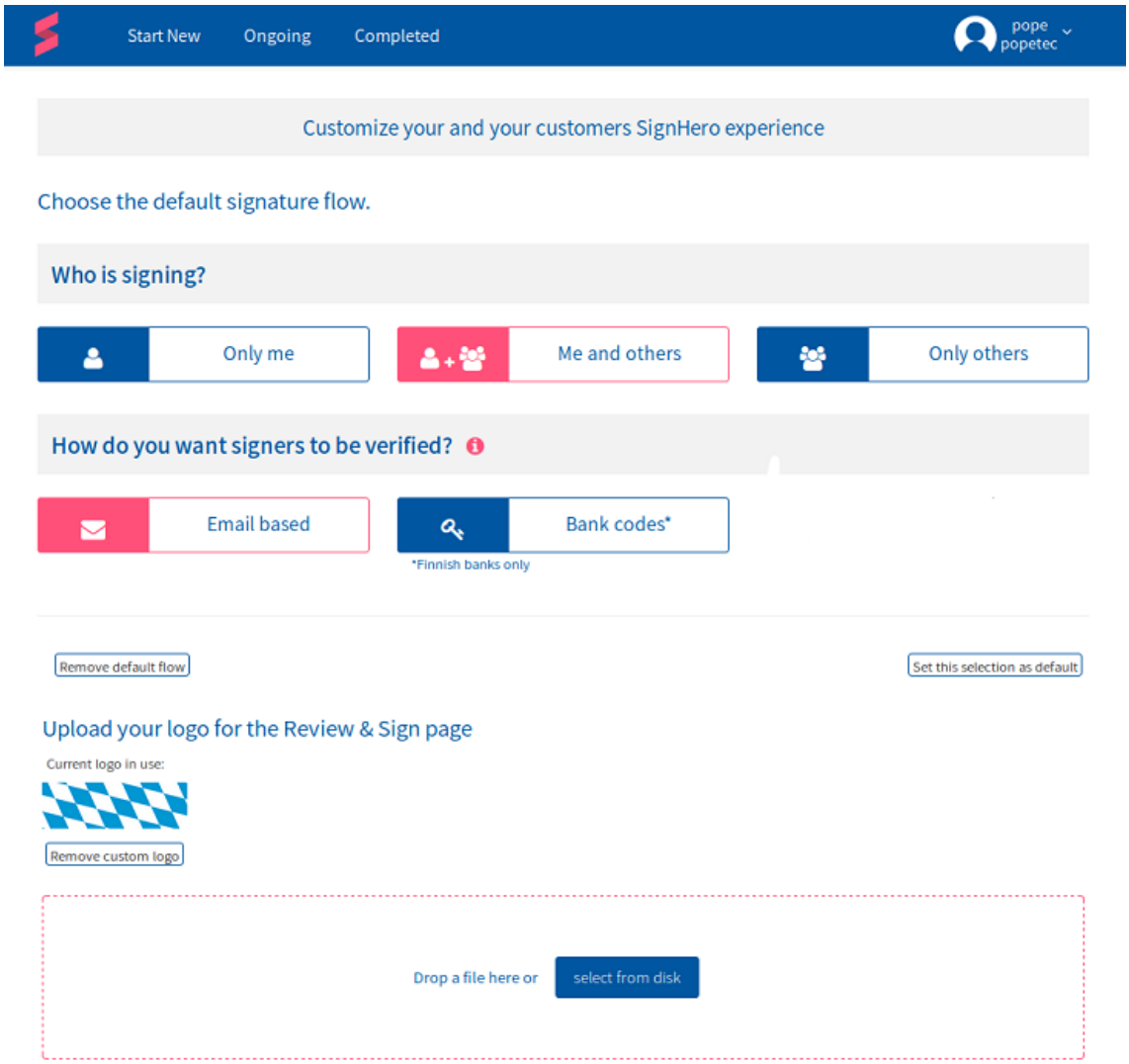

Figure 6.3. Customization pane of the configuration management tool.

In the Figure 6.3 is shown the layout of customization pane as it is shown after user has updated the organization's configurations. The upper section consists of workflow management. The WorkflowManager widget is actually an extension of the original widget used in the sign flow initialization page, but with some features pruned out. Remove default and save default buttons calls methods from ConfigLib that eventually modifies the database. The lower part of the page is for logo management. The LogoManager widget shows the current organization logo as it is shown both in the standalone signing page and in the emails. Usability, which is extremely important in user-driven management tools, is not forgotten on this page: some extra animations are implemented to give the user a sense of progress and feedback, such as fade in and fade out animations when updating the organization logo.

6.5 Evaluation

The developed prototype of a configuration management tool is a success in a sense that it has support for the required variability points. Updating, deleting and retrieving configuration data can be done with the tool, and the chosen approach with Usergrid collections, entities and properties makes it easy to add new variability points. Extending the API with new configuration possibilities can be done without too much effort. After all, the main idea was to keep things simple: focus on configuration instead of customization. Users also benefit from simple, easy-to-use application that does the required with minimum effort and does not confuse the user with too many options.

Big share of the overall work was actually spent on the background in order to minimize the technical debt. Making a customizable cloud-native application is a complex task, therefore spending time on the design and understanding the requirements and even limitation will pay off in the future as extending the application is much easier. It can be said that the current prototype fosters the cloud characteristics to some extent, for example the application does not store application state on the same server which is the prerequisite for scalability. However, creating a scalable application was out of scope for which caching was also left out.

In addition to implementing the configuration management tool itself, there was also a need to extend the SignHero main service in order to apply the tenant configurations. This was done successfully with tenant-isolated components, but some of the decisions related to the design turned out be non-optimal, mainly because code usability was overlooked. Most notably giving the responsibility to load and inject both the signing page logo and signing flow configurations in the client-side is rather dubious. This approach was selected because it was faster to implement by using the same API that was developed for the configuration management tool. Even though this route is simpler than embedding the tenant configurations in the backend to the requested module, it can be a nuisance to load the page and its customized parts asynchronously in small parts. It would also require proper notifications for the user such as a loading bar or transitions after successful or erroneous cases. These usability features were not implemented.

Uploading a customized logo and saving the default workflow are actually two different types of tenant configurations. Former is a more common type and it is used to replace something on the application for good, while the latter automatically uses the provided values while the part of the application itself is the same for each tenant. The first one could be done fully in the backend, and the second one naturally relies also on client-side logic. Shifting most of the logic to the backend is also dependent of the initial design of the main service. In other words, re-engineering the application to support configurations and customizations has to be taken into account in time and cost estimations.

MVS of the SignHero did came out during the implementation phase. The architecture and the design remained mostly the same with only minor modifications. Connectivity to Usergrid was updated slightly, tenant identifier was switched for another Usergrid property and UI faced some updates which had also an impact on the configuration management tool.

6.6 Improvement possibilities in the future

It is important to note that the developed tool is still nothing but a prototype. This leaves many opportunities available about which actions should be taken for the future versions. First of all, now that the foundation has been created for a full-scale configuration management tool, next step is to continue the work until the tool is in a finalized, production ready form. After the basic development work, including switching the asset storage to something else than a filesystem storage, cloud characteristics and multi-tenant application aspects shall be improved: most importantly application's scalability has to be implemented, customizations can be considered to be taken into use and creating means to control and monitor QoS has to be done.

Cloud applications should have rapid elasticity built in to the system. Scalability should be supported by adding and removing instances on the fly, and usually by an automatic procedure. But for this tool, the workload could be more easily minimized with a cache. Caching is an efficient way to hasten an application by reducing database queries and disk I/O, which would make cache – and how to embed into the tool – a sound topic for next research. Later on, more application instances and database servers could be added in order to handle all the incoming requests, if cache as such is not sufficient anymore. The developed tool is stateless so it does not store application state or session data on the same server, so the tool can already work in an environment where there is a load balancer routing requests to whichever instance. Additionally, if one day the configuration possibilities are considered as a vital part for the business, then it would be logical to split the main service and the configuration management tool into separate instances. The growing amount of requests related to tenant configurations or even customizations would make a hefty increase in the workload which might risk the QoS within the whole service. Separating the services would require some sort of a message queue and more attention to the APIs.

At some point SLA could be defined differently for Premium and Freemium users. For example Premium users could be offered a better QoS and separate “SignHero-Customization” instances for increased availability, whereas Freemium users with a lesser QoS would be served with a single server for both configurations and the main service. Considering performance-isolation, there has to be active monitoring of the used resources to make sure that a single user is not able to cause decline in the service quality for every user, as seen with the Salesforce example. Components processing

configuration requests should have e.g. a timeout possibility if executing takes too long, or each tenant has a pre-defined maximum rate for resource utilization.

The implemented configuration API and the database structure are flexible enough to be extended with more variability points. However, the increased complexity might bring forth a need for a “preview mode” for selected configurations if the user can choose from a wide range of features and options. Getting rid of “trial-and-error” and easing tenant's work can be considered as a requirement of the tenant-driven approach. Walraven et al. (2014) stated that handling all the variations and their parameters can become difficult when the amount is increased. When examining email templates from this point of view, there is a risk that the current approach will not be sufficient. Currently the email templates behave so that only the image source is substituted with the specific tenant version, but in the future, there could be customization for, e.g., text content and footer. There is already a dozen of different emails which, combined with language versions, eventually results in a significant amount of different template variations. In this case, it might be worthwhile to consider using dedicated components instead of tenant-isolated components. Each ready-to-use template injected with tenant parameters could be stored to the asset storage, instead of having every parameter of each template stored to Cassandra. This would make retrieving template easier but with the cost of storage space and lower resource sharing utilization rate. Users could still use the configuration management tool with predefined variability points, but the backend would generate complete files based on the input and process the templates as if they were uploaded as such by tenant. Input sanitation and validation would be necessary in this kind of solution. Later on the tenant could be allowed to actually upload the custom template, but it would put more pressure to provide more support for tenants, thus increasing the operational costs.

7. CONCLUSION

Goal of the thesis was to examine the requirements of multi-tenant applications, and make the new digital signature cloud service called SignHero configurable by its users. The work involved creating a configuration management tool that supports the given customization objectives: uploading a new logo to be used in the SaaS and saving a default workflow for making the signing process faster. The uploaded organization logo substitutes the default logo in the signing process and emails, and the saved default workflow is applied automatically when the user starts a new signing process. In addition to the developed tool, the main service had to be extended in a way that the user's configurations are put to use.

The resulted tool follows the main principles of MTAs. The configuration management tool is kept simple while preferring configuration over customization. Self-service tools are required to be easy-to-use, but the role of usability turned out to be more important than initially thought. New variability points can be easily added to the application due to the flexible JSON data format in the chosen persistence solution called Usergrid. Working on the prototype also proved that when making a multi-tenant application, it is rather pointless to just apply the customization features on top of the application: the features have to be naturally included in the application design, which unfortunately increases the engineering costs and code complexity.

The work included a research on cloud applications from a multi-tenancy point of view. Cloud as an environment is difficult for developers due to added complexity caused by the essential cloud characteristics: access via network, on-demand self-service, pay-per-use, resource pooling and rapid elasticity. Nowadays cloud and multi-tenant applications are trying to supersede the traditional single-tenant and ASP models, because in overall multi-tenant systems benefit from higher resource utilization and significantly lower operational costs. It became clear that multi-tenancy can be considered as one of the main aspects of cloud computing due to leveraging economies of scale. It is important to note that there are also alternatives, such as middleware or OS-level based multi-tenancy, for making customizable systems for tenants. Creating an MTA might not always be necessary, and single-tenant approach can also be viable. SaaS developers have to take into account the engineering costs, the estimated amount of tenants, and the scope of the customization features. Decision makers can, for example, utilize cost models such as the presented one in Section 2.4.

First steps towards a customizable multi-tenant application were taken. Future work after finalizing the prototype and deploying to the production will include improving the configurability options and cloud characteristics in the application, most notably scalability and measured service. Scalability can be easily enhanced by taking a cache into

use. Performance-isolation and assuring the promised QoS for tenants are other future research topics.

REFERENCES

Antapoulos, N. & Gillam, L., 2010. Cloud Computing: Principles, Systems and Applications. Springer. 379 p.

Ashalatha, R. & Jayashree, A., 2016. Multi Tenancy Issues in Cloud Computing for SaaS environment. IEEE International Conference on Circuit, Power and Computing Technologies [ICCPCT]. pp. 1-4.

Avaintec, 2017. Avaintec - Let's make data work. Accessed 22 August 2017. Available: http://www.avaintec.com

Baun, C., Kunze, M., Nimis, J. & Tai, S., 2011. Cloud Computing: Web-Based Dynamic IT Services. Springer.

Bezemer, C.-P. & Zaidman, A., 2010. Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE). pp. 88-92.

Bitbucket, 2017. Bitbucket Server and Data Center | Bitbucket. Accessed 22 August 2017. Available: https://bitbucket.org/product/enterprise

Buyya, R., Broberg, J. & Goscinski, A. M., 2011. Cloud Computing: Principles and Paradigms. Wiley Publishing. 664 p.

Cassandra, 2017. Apache Cassandra. Accessed 5 August 2017. Available: http://cassandra.apache.org/

Chong, F. & Carraro, G., 2006. Architecture strategies for catching the long tail. Technical report. Accessed 12 February 2017. Available: http://msdn.microsoft.com/en-us/library/aa479069.aspx.

Chong, F., Carraro, G. & Wolter, R., 2006. Multi-Tenant Data Architecture. Accessed 18 August 2017. Available: https://msdn.microsoft.com/en-us/library/aa479086.aspx.

Coleman, M., 2016. Containers are not VMs. Accessed 22 August 2017. Available: https://blog.docker.com/2016/03/containers-are-not-vms/.

Dojo, 2017. Dojo Toolkit. Accessed 30 July 2017. Available : https://dojotoolkit.org/

Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P. 2014. Cloud Computing Patterns: Fundamentals to Design, Build and Manage Cloud Applications. Springer. 393 p.

Fink, G. & Flatow, I., 2014. Pro Single Page Application Development: Using Backbone.js and ASP.net. Apress. 324 p.

Gey, F., Van Landuyt, D. & Joosen, W., 2015. Middleware for Customizable Multi-Staged Upgrades of Multi-Tenant SaaS Application. In: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC). IEEE. pp. 102-111.

GraphQL, 2017. A query language for your API. Accessed 22 August 2017. Available: http://graphql.org/

Guo, C. J., Sun, W., Huang, Y., Wang, Z. H., Gao, B. 2007. A Framework for Native Multi-Tenancy Application Development and Management. In: E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on . IEEE. pp. 551-558.

Johanan, J., 2014. Building Scalable Apps with Redis and Node.js. Packt Publishing Ltd. 293 p.

Kabbedijk, J., Bezemer, C.-P., Jansen, S. & Zaidman, A., 2015. Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. In: Journal of Systems and Software. Elsevier Science Inc. pp. 139-148.

Kabbedijk, J. & Jansen, S., 2011. Variability in Multi-tenant Environments: Architectural Design Patterns from Industry. In: Advances in Conceptual Modeling. Recent Developments and New Directions: ER 2011. Springer, pp. 151-160.

Kavis, M. J., 2014. Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS). Wiley Publishing. 224 p.

Krebs, R., Momm, C. & Kounev, S., 2012. Architectural Concern in Multi-Tenant SaaS Applications. In: Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER2012).

Lee, W. & Choi, M., 2012. A Multi-tenant Web Application Framework for SaaS. In: 2012 IEEE Fifth International Conference on Cloud Computing. IEEE, pp. 970-971.

Marinescu, D. C., 2013. Cloud Computing: Theory and Practice. Morgan Kaufmann.

Mell, P. & Grance, T., 2011. SP 800-145. The NIST Definition of Cloud Computing. National Institute of Standards and Technology.

Microsoft, 2017. Azure Stack - Hybrid Cloud | Microsoft Azure. Accessed 22 August 2017. Available: https://azure.microsoft.com/en-us/overview/azure-stack/

Mikowski, M. S., 2015. Replace RESTful APIs with JSON-Pure – Michael S. Mikowski – SPA (UI/UX/server) architect and author. Accessed 17 July 2017. Available: https://mmikowski.github.io/json-pure/

Momm, C. & Krebs, R., 2011. A Qualitative Discussion of Different Approaches for Implementing Multi-Tenant SaaS Offerings. In: Software Engineering 2011. pp. 131-150.

Nigam, S., 2015. Co$t Effective Cloud Application Architectures. EMC. 46 p.

Node.js, 2017. About | Node.js. Accessed 29 July 2017. Available: https://nodejs.org/en/about/

npm, 2017. npm. Accessed 29 July 2017. Available: https://www.npmjs.com/.

Ochei, L. C., Bass, J. M. & Petrovski, A., 2015. Evaluating Degrees of Multitenancy Isolation: A Case Study of Cloud-Hosted GSD Tools. In: 2015 International Conference on Cloud and Autonomic Computing. pp. 101-112.

Omote, Y., Shinagawa, T. & Kato, K., 2015. Improving Agility and Elasticity in Bare-metal Clouds. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM. pp. 145-159.

OpenStack, 2017. OpenStack Open Source Cloud Computing Software. Accessed 22 August 2017. Available: https://www.openstack.org/

Rountree, D. & Castrillo, I., 2013. The Basics of Cloud Computing : Understanding the Fundamentals of Cloud Computing in Theory and Practice. Syngress Publishing.

Saleh, A. I., Fouad, M. A. & Abu-Elkheir, M., 2014. Classifying Requirements for Variability Optimization in Multi-tenant Applications. In: 2014 IEEE 6th International Conference on Cloud Computing Technology and Science. IEEE. pp. 32-37.

Salesforce.com, 2014. An Introduction to VisualForce. Accessed 27 May 2017. Available: https://developer.salesforce.com/page/An_Introduction_to_Visualforce.

Salesforce.com, 2016. Multi Tenant Architecture. Accessed 27 May 2017. Available: https://developer.salesforce.com/page/Multi_Tenant_Architecture.

Salesforce.com, 2017a. Understand the Salesforce Architecture Unit. Accessed 27 May 2017. Available:

https://trailhead.salesforce.com/en/modules/starting_force_com/units/starting_understanding_arch

Salesforce.com, 2017b. What is Apex?. Accessed 27 May 2017. Available: https://developer.salesforce.com/docs/atlas.enus.apexcode.meta/apexcode/apex_intro_what_is_apex.htm.

Salesforce.com, 2017c. What is salesforce. Accessed 27 May 2017. Available at: https://www.salesforce.com/eu/crm/what-is-salesforce/

Saraswathi, M. & Bhuvaneswari, T., 2014. Multitenant SaaS Model of Cloud Computing: Issues and Solutions. In: 2014 International Conference on Communication and Network Technologies. IEEE. pp. 27-32.

Saxena, U., Sachdeva, S. & Batra, S., 2015. Moving from Relational Data Storage to Decentralized Structured Storage System. In: Databases in Networked Information Systems: 10th International Workshop. Springer, pp. 180-194.

Sharda, R., Chaudbury, M., Rajesh, P. & Srinivasa, G., 2014. Patterns of Multi-tenant SaaS Applications. EMC. 42 p.

Tsai, W.-T. & Sun, X., 2013. SaaS Multi-Tenant Application Customization. In: 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering. IEE. pp. 1-12.

Usergrid, 2017. Apache Usergrid - The BaaS not made for Hipsters. Accessed 5 August 2017. Available: http://usergrid.apache.org

W3, 2009. W3 Document Object Model. Accessed 21 August 2017. Available: https://www.w3.org/DOM/

Walraven, S., Truyen, E. & Joosen, W., 2011. A Middleware Layer for Flexible Cost-Efficient Multi-tenant Applications. In: Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware. Springer. pp. 370-389.

Walraven, S., Van Landuyt, D., Tryuen, E. & Handekyn, K. J. W., 2014. Efficient customization of multi-tenant Software-as-a-Service applications with service lines. In: Journal of Systems and Software. CrossMark. pp. 48-62.

Sun, W., Zhang, X., Guo, C. J., Sun, P., Su, H. 2008. Software as a Service: Configuration and Customization Perspectives. In: 2008 IEEE Congress on Services Part II. IEEE. pp. 18-25.